

University of Crete
Computer Science Department

INTEGRATED METAPROGRAMMING SYSTEMS: LANGUAGE, TOOLS AND PRACTICES

by
YANNIS LILIS

In partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Heraklion, November 2013

The work reported in this thesis has been conducted at the Human Computer Interaction (HCI) laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology - Hellas (FORTH), and has been financially supported by a FORTH-ICS scholarship.

University Of Crete
Computer Science Department

INTEGRATED METAPROGRAMMING SYSTEMS: LANGUAGE, TOOLS AND PRACTICES

by
YANNIS LILIS

A thesis submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

Author: _____
Yannis Lilis, University of Crete

Examination Committee:

Supervisor _____
Anthony Savidis, Professor, University of Crete

Member _____
Evangelos Markatos, Professor, University of Crete

Member _____
Dimitris Plexousakis, Professor, University of Crete

Member _____
Paris Avgeriou, Professor, University of Groningen, Holland

Member _____
Yannis Smaragdakis, Associate Professor, University of Athens

Member _____
Spyros Lalas, Associate Professor, University of Thessaly

Member _____
Nikolaos Papaspyrou, Associate Professor, National Technical
University of Athens

Approved by: _____
Angelos Bilas, Professor, University of Crete
Chairman of the Graduate Studies Committee

Heraklion, November 2013

Integrated Metaprogramming Systems: Language, Tools and Practices

YANNIS LILIS

PhD Thesis

University Of Crete
Computer Science Department

Abstract

Metaprogramming is an advanced language feature enabling to mix programs with definitions that generate source code to be put in their place. Such definitions are called *metaprograms* and are executed during the translation of the main program. While metaprograms are essentially programs they are mostly treated as special cases without sharing the current programming practices and development tools. In this context, we emphasize the need for a methodological integration between metaprograms and normal programs presenting a systematic proposition towards *integrated metaprogramming systems*. We cover and implement aspects related to language, programming model, tool support and deployment practices.

We identify a set of primary requirements related to language implementation, metaprogramming features, software engineering support, and programming environments, that are necessary to achieve such integration and elaborate on addressing them in the implementation of a metaprogramming system. In particular, we introduce the notion of *integrated metaprograms*, as coherent programs assembled from specific meta-code fragments present in the source code. We prove the expressiveness of this programming model and illustrate its software engineering advantages through case studies that reflect demanding scenarios of exception handling, design patterns and design by contract. Then we present an integrated tool-chain that treats metaprograms as first-class citizens of the programming environment, incorporating them into the workspace management and supporting them with a full-scale build process. We also elaborate on the way we provide precise compile-error reporting and full-power source-level debugging facilities for metaprograms.

Regarding model integration, we show how Aspect-Oriented Programming (AOP), a paradigm originally considered only for normal programs, is effectively extended and applied in a metaprogramming context. In particular, we present a systematic proposition for introducing aspect orientation in the entire processing pipeline of a metaprogramming system. Additionally, we discuss an implementation approach treating aspects as batches of transformation metaprograms, the latter deploying a custom AOP-related library we offer. Example scenarios are discussed demonstrating how the proposed aspect system is used in practice, while we present how full-scale source-level aspect debugging is facilitated during the program compilation process.

Finally, we propose deployment practices that utilize metaprogramming to achieve reusability at a macroscopic scale. In this direction, we present a methodology for implementing reusable design patterns and exception handling templates by realizing them as metaprogram libraries that can be deployed on demand. We also discuss an improved Model-Driven Engineering (MDE) practice where the outcomes of MDE-tools become read-only Abstract Syntax Trees (ASTs) instead of source code to resolve the inherent maintenance issues in such tools. In our approach the application source code involves metaprogramming to deploy and manipulate the generated code fragments as ASTs, instead of being built around the generated code with custom modifications and extensions.

Ολοκληρωμένα Συστήματα Μεταπρογραμματισμού: Γλώσσα, Εργαλεία και Πρακτικές

ΙΩΑΝΝΗΣ ΛΙΛΗΣ

Διδακτορική Διατριβή

Πανεπιστήμιο Κρήτης
Τμήμα Επιστήμης Υπολογιστών

Περίληψη

Ο μεταπρογραμματισμός είναι ένα προηγμένο χαρακτηριστικό γλωσσών που επιτρέπει στα προγράμματα να αναμιγνύονται με ορισμούς που παράγουν κώδικα για να μπει στη θέση τους. Αυτοί οι ορισμοί ονομάζονται *μεταπρογράμματα* και εκτελούνται κατά τη διάρκεια της μετάφρασης του κυρίως προγράμματος. Παρότι τα μεταπρογράμματα είναι ουσιαστικά προγράμματα, συχνά αντιμετωπίζονται ως ειδικές περιπτώσεις, χωρίς να μοιράζονται τις τρέχουσες προγραμματιστικές πρακτικές και τα εργαλεία ανάπτυξης. Σε αυτό το πλαίσιο, τονίζουμε την ανάγκη της μεθοδολογικής ενοποίησης των μεταπρογραμμάτων και των κανονικών προγραμμάτων μέσω μιας συστηματικής πρότασης για *ολοκληρωμένα συστήματα μεταπρογραμματισμού*. Ειδικότερα, καλύπτουμε και υλοποιούμε πτυχές της γλώσσας, του μοντέλου προγραμματισμού, της υποστήριξης εργαλείων και των πρακτικών ανάπτυξης.

Εντοπίζουμε ένα σύνολο βασικών απαιτήσεων που σχετίζονται με την υλοποίηση της γλώσσα, τα χαρακτηριστικά του μεταπρογραμματισμού, την υποστήριξη της παραγωγής λογισμικού, και τα περιβάλλοντα προγραμματισμού, οι οποίες είναι απαραίτητες για την επίτευξη αυτής της ενοποίησης και παρέχουμε λεπτομέρειες για την αντιμετώπισή τους στην υλοποίηση ενός συστήματος μεταπρογραμματισμού. Ειδικότερα, εισάγουμε την έννοια των *ενοποιημένων μεταπρογραμμάτων*, ως συνεκτικά προγράμματα που συναρμολογούνται από συγκεκριμένα τμήματα μετακώδικα που βρίσκονται μέσα στον πηγαίο κώδικα. Αποδεικνύουμε την εκφραστικότητα αυτού του προγραμματιστικού μοντέλου και παρουσιάζουμε τα πλεονεκτήματα του ως προς την ανάπτυξη λογισμικού μέσω παραδειγμάτων που αντικατοπτρίζουν απαιτητικά σενάρια σχετικά με χειρισμό εξαιρέσεων, σχεδιαστικά πρότυπα και σχεδίαση βασισμένη σε συμβόλαιο. Στη συνέχεια, παρουσιάζουμε μια

ολοκληρωμένη σειρά εργαλείων που αντιμετωπίζουν τα μεταπρογράμματα ως βασικές οντότητες ενός περιβάλλοντος προγραμματισμού, ενσωματώνοντάς τα στη διαχείριση του χώρου εργασίας και υποστηρίζοντάς τα με μια διαδικασία μεταγλώττισης πλήρους κλίμακας. Επίσης αναλύουμε τον τρόπο με τον οποίο παρέχουμε ακριβείς αναφορές για λάθη μεταγλώττισης καθώς και πλήρως λειτουργική εκσφαλμάτωση πηγαίου κώδικα για μεταπρογράμματα.

Σχετικά με την ενοποίηση του μοντέλου, δείχνουμε πώς ο προγραμματισμός βασισμένος σε προοπτικές (Aspect-Oriented Programming), ένα προγραμματιστικό παράδειγμα που αρχικά υφίσταντο μόνο για κανονικά προγράμματα, μπορεί να επεκταθεί και να εφαρμοστεί στο πλαίσιο του μεταπρογραμματισμού. Συγκεκριμένα, παρουσιάζουμε μια συστηματική πρόταση για την εισαγωγή προοπτικών σε όλα τα στάδια της διαδικασίας μεταγλώττισης σε ένα συστήματα μεταπρογραμματισμού. Επιπρόσθετα, αναλύουμε μια μέθοδο υλοποίησης που εφαρμόζει τις προοπτικές ως παρτίδες μετασχηματιστικών μεταπρογραμμάτων, τα οποία χρησιμοποιούν μια ειδική βιβλιοθήκη για προοπτικές την οποία παρέχουμε. Συζητάμε πρακτικά σενάρια χρήσης για την προτεινόμενη πρακτική προοπτικών, ενώ παρουσιάζουμε τον τρόπο με τον οποίο η πλήρους κλίμακας εκσφαλμάτωση των προοπτικών σε επίπεδο πηγαίου κώδικα μπορεί να υποστηριχθεί κατά τη διάρκεια της μεταγλώττισης.

Τέλος, προτείνουμε πρακτικές που χρησιμοποιούν μεταπρογραμματισμό για την επίτευξη επαναχρησιμοποίησης σε μακροσκοπική κλίμακα. Σε αυτή την κατεύθυνση, παρουσιάζουμε μια μεθοδολογία για την υλοποίηση επαναχρησιμοποιήσιμων σχεδιαστικών προτύπων και καλουπιών χειρισμού εξαιρέσεων, πραγματοποιώντας τα ως βιβλιοθήκες μεταπρογραμματισμού που μπορούν να χρησιμοποιηθούν κατά περίπτωση. Επιπλέον, περιγράφουμε μια βελτιωμένη πρακτική για την ανάπτυξη λογισμικού που βασίζεται σε μοντέλα (Model-Driven Engineering) όπου οι έξοδοι των εργαλείων μοντελοποίησης μετατρέπονται σε αφαιρετικά συντακτικά δέντρα (ASTs) που είναι μόνο για ανάγνωση αντί για πηγαίο κώδικα, στοχεύοντας στην επίλυση των εγγενών προβλημάτων συντήρησης αυτών των εργαλείων. Στην προσέγγισή μας, ο πηγαίος κώδικας της εφαρμογής χρησιμοποιεί μεταπρογραμματισμό για να δημιουργήσει και να διαχειριστεί τα παραγόμενα τμήματα κώδικα ως αφαιρετικά συντακτικά δέντρα, αντί να χτίζεται πάνω στον παραγόμενο κώδικα με διάφορες τροποποιήσεις και επεκτάσεις.

Ευχαριστίες (Acknowledgements)

Θα ήθελα να ευχαριστήσω ιδιαίτερα τον επόπτη, μου Καθηγητή του Τμήματος Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης, κ. Αντώνη Σαββίδη, για την συνεχή καθοδήγηση και υποστήριξή του στο πλαίσιο της συνεργασίας μας όλα τα χρόνια στο Εργαστήριο Αλληλεπίδρασης Ανθρώπου-Υπολογιστή, του Ινστιτούτου Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας, ειδικότερα στο πλαίσιο της εκπόνησης της διδακτορικής μου διατριβής.

Θα ήθελα επίσης να ευχαριστήσω τα μέλη της τριμελούς επιτροπής της διδακτορικής μου διατριβής, κ. Δημήτρη Πλεξουσάκη και κ. Βαγγέλη Μαρκάτο, Καθηγητές του Τμήματος Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης, για τις εποικοδομητικές παρατηρήσεις και σχόλια που έκαναν κατά τα πρώτα στάδια αυτής της εργασίας.

Επιπλέον, ευχαριστώ το Γιάννη Βαλσαμάκη για τη συνεργασία μας στο κομμάτι του συνδυασμού του μεταπρογραμματισμού με το Model-Driven Engineering, αντικείμενο που αποτέλεσε κοινή αφετηρία για τη δικιά του μεταπτυχιακή εργασία και μέρος της δικιάς μου διδακτορικής διατριβής.

Τέλος, θέλω να ευχαριστήσω τη γυναίκα μου, Έφη, του γονείς μου, Βαγγέλη και Δήμητρα, τον αδερφό μου, Γιώργο, και όλους τους φίλους μου για τη βοήθεια και τη στήριξη που μου παρείχαν όλα αυτά τα χρόνια.

List of Publications

- Lilis, Y., Savidis, A. 2013. *An Integrated Implementation Framework for Compile-Time Metaprogramming*. Softw: Pract. Exper. Available at: <http://dx.doi.org/10.1002/spe.2241>.
- Lilis, Y., Savidis, A. *Aspects for Stages: Cross Cutting Concerns for Metaprograms*. Journal of Object Technology. To appear.
- Lilis, Y., Savidis, A. 2013. *An Integrated Approach to Source Level Debugging and Compile Error Reporting in Metaprograms*. Journal of Object Technology 12(3): 1: pp. 1-26. Available at: <http://dx.doi.org/10.5381/jot.2013.12.3.a2>.
- Lilis, Y., Savidis, A., Valsamakis, Y. 2014. *Staged Model-Driven Generators: Shifting Responsibility for Code Emission to Embedded Metaprograms*. In Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014), 7-9 January, Lisbon, Portugal. To appear.
- Lilis, Y., Savidis, A., Valsamakis, Y. 2013. *Self Model-Driven Engineering Through Metaprograms*. In Proceedings of the 17th Panhellenic Conference on Informatics (PCI 2013), 19-21 September. ACM, New York, NY, USA, pp. 136-143. Available at: <http://dx.doi.org/10.1145/2491845.2491872>.
- Lilis, Y., Savidis, A. 2012. *Implementing Reusable Exception Handling Patterns with Compile-Time Metaprogramming*. In SERENE 2012, 4th International Workshop on Software Engineering for Resilient Systems, September 27-28, Pisa, Italy, Springer LNCS 7527, pp. 1-15. Available at: http://dx.doi.org/10.1007/978-3-642-33176-3_1
- Lilis, Y., Savidis, A. 2012. *Supporting Compile-Time Debugging and Precise Error Reporting in Meta-Programs*. In TOOLS 2012, International Conference on Objects, Models, Components, Patterns, 29-31 May, Prague, Czech Republic, Springer LNCS 7304, pp. 155-170. Available at: http://dx.doi.org/10.1007/978-3-642-30561-0_12.

Table of contents

Abstract	iii
Περίληψη	v
Ευχαριστίες (Acknowledgements).....	vii
List of Publications	viii
Table of contents	ix
List of Figures	xiv
List of Tables	xix
1 Introduction.....	1
1.1 Motivation	5
1.2 Contributions	7
1.3 Outline	9
2 Related Work	10
2.1 Background Information	10
2.2 Language Support for Metaprogramming	11
2.2.1 Macro Systems	11
2.2.1.1 Text-Based	12
2.2.1.2 Syntax-Based	14
2.2.2 Multi-Stage Languages	16
2.2.3 Runtime Metaprogramming	20
2.2.4 Compile-Time Metaprogramming	22
2.2.5 Modifiable Syntax and Semantics.....	28
2.3 Tool Support for Metaprogramming	33
2.3.1 Error Reporting	33
2.3.2 Debugging	34
2.3.3 IntelliSense	37
2.3.4 Browsing	38

3	Requirements	40
3.1	Analysis	40
3.1.1	Exploiting Normal Language Features and Tools.....	40
3.1.2	Supporting Context-Free and Context-Sensitive Generation.....	42
3.1.3	Composing and Generating All Language Constructs.....	43
3.1.4	Sharing and Separation of Concerns among Stages and Main.....	45
3.1.5	Programming Model for Stages Equal to Normal Programs	46
3.1.6	Treating Stages as First-Class Citizens of the IDE	49
3.1.6.1	Source Browsing and Editing Automations.....	49
3.1.6.2	Build Tools.....	50
3.2	Review.....	52
3.2.1	Compile-Time Metaprogramming	53
3.2.1.1	C++	53
3.2.1.2	Template Haskell	53
3.2.1.3	Nemerle	54
3.2.1.4	Converge	55
3.2.1.5	MetaLua	56
3.2.1.6	Groovy	57
3.2.2	Runtime Metaprogramming	57
3.2.2.1	Lisp and Scheme	57
3.2.2.2	MetaML, MetaOCaml and Mint	59
4	Metalanguage.....	60
4.1	Integrated Model	60
4.2	Syntax and Semantics.....	63
4.2.1	AST Tags.....	63
4.2.2	Staging Tags	67
4.2.3	Stage Assembly and Evaluation.....	71

4.2.4	Enabling Metagenerators.....	75
4.2.5	Context Sensitivity	76
4.2.6	Metacode Libraries.....	77
4.3	Expressiveness.....	78
4.4	Discussion	80
4.5	Case Studies	81
4.5.1	Exception Handling.....	82
4.5.2	Design Patterns.....	83
4.5.3	Design By Contract	88
5	Tool Extensions	91
5.1	Workspace Manager.....	91
5.2	Build System	94
5.3	Compile-Error Reporting	97
5.3.1	Compile-Error Chain across Stages and their Outputs	98
5.3.2	Study for Runtime Metaprogramming	100
5.4	Source-Level Debugging.....	102
5.4.1	Translation-Time Debug Session	103
5.4.2	Supporting Stage Breakpoints.....	105
5.4.3	Enabling AST Inspection	107
5.4.4	Study for Runtime Metaprogramming	109
5.4.4.1	Stage Debugging.....	109
5.4.4.2	Stage Breakpoints	112
5.4.4.3	Multi-Stage Languages	114
6	Support for Aspects	116
6.1	Aspects for Stages	118
6.1.1	Weaving Options.....	119
6.1.2	Aspect Categories.....	123

6.2	Aspects without Dedicated Languages.....	127
6.2.1	Aspects as Transformation Batches	127
6.2.2	Aspect Transformation Library	130
6.2.3	Aspects in the Workspace Manager	132
6.2.4	Aspects in the Build Process	133
6.2.5	Debugging Aspects	135
6.2.5.1	Reviewing Woven Code	136
6.2.5.2	Providing Accurate Compile Errors.....	137
6.2.5.3	Tracing the Evaluation of Aspects	138
6.3	Case Studies	140
6.3.1	Aspects to Insert Staging.....	140
6.3.2	Aspects for Custom Static Analysis	143
6.3.3	Aspects to Introduce Memoization in Stages	144
6.3.4	Aspects for Tracing Diagnostics in Stages.....	145
6.3.5	Aspects for Locking Shared Objects in Stages	145
6.3.6	Aspects for Exception Handling in Stages	146
6.3.7	Aspects for Decorating Classes in Stages	146
6.3.8	Aspects for Custom AST Iteration in Stages	147
6.3.9	Aspects for AST Validation in Stages.....	148
6.4	Discussion	149
6.5	Comparison to Current AOP Practices.....	151
7	Advanced Practices.....	153
7.1	Design Pattern Generators.....	153
7.1.1	Decorator.....	155
7.1.2	Adapter	156
7.1.3	Flyweight.....	158
7.1.4	Undo-Redo	160

7.2	Exception Handling Templates	163
7.2.1	Exception Handling Scenarios	165
7.2.2	Exception Policies	168
7.2.3	Process Modeling Patterns	170
7.2.4	Pattern Combinations	172
7.2.5	Comparison to AOP	173
7.3	Staged Model-Driven Generators.....	175
7.3.1	MDE Maintenance Issues.....	175
7.3.2	Improving the MDE Process	177
7.3.2.1	Deploying Staging.....	178
7.3.2.2	Deploying Reflection	179
7.3.3	Self MDE Deployment.....	180
7.3.4	Case Studies	182
7.3.4.1	User-Interface Generation.....	183
7.3.4.2	Class Hierarchy Generation	184
7.3.4.3	Combining Multiple MDE Tools.....	186
8	Conclusions and Future Work	188
8.1	Summary	188
8.2	Conclusions	190
8.3	Future Work	193
	Bibliography	195

List of Figures

Figure 1.1 – Abstract view of the metaprogramming process	2
Figure 2.1 – Processing diagram for macro systems	12
Figure 2.2 – General staging process in multi-stage languages.....	17
Figure 2.3 – Runtime code generation and execution through reflection: C# (top) and Java (bottom).	21
Figure 3.1 – Context-free (top) versus context-sensitive (bottom) code generation ...	42
Figure 3.2 – Common evaluation of stages in popular multi-stage languages (e.g. MetaML, MetaOCaml, Converge, Metalua, etc.) and macro systems (e.g. Lisp): inside-out for nested stages, and top-down for top level stages, all as independent execution sessions. Dotted lines connect stage fragments of the same nesting level whose concatenation could comprise a single stage program.	46
Figure 3.3 – Illustrating the support of IntelliSense information for both metaprograms (left) and their outcomes (right) when deploying CPP for code composition.....	50
Figure 4.1 – Concept of integrated metaprograms: (i) comprising all stage fragments at the same nesting in their order of appearance; (ii) denoting a sequential control flow among stage fragments; and (iii) providing scope visibility to previous stage fragments.....	61
Figure 4.2 – Typical evaluation order in multi-stage languages - $f_{i,j}$ are staged expressions with i enumerating staged code blocks and j denoting the stage nesting in the respective block.....	63
Figure 4.3 – Without supporting <i>define</i> all stage snippets are by default collected inside the main stage block, that could cause ill-formed C++ syntax as C++ forbids local function definitions (shaded code).	70
Figure 4.4 – When supporting <i>define</i> , only stage code from <i>execute</i> directives is collected inside the main stage block; code from <i>define</i> is assembled with rest non-	

stage global definitions, following their order of appearance, resulting in syntactically correct C++ code.....	70
Figure 4.5 – The automatic treatment of <i>execute</i> directives involving global definitions as <i>define</i> directives disables encapsulation and information hiding for element categories allowed both at global and local scope; this may cause semantic errors, such as replicate definitions (name conflicts).....	71
Figure 4.6 – Illustrating the assembly of integrated metaprograms with a general example with arrows outlining dependencies – only the required definitions from the main program are included.	72
Figure 4.7 – Left: main with its intermediate and final versions; Right: Integrated metaprograms from the original and intermediate main versions.	74
Figure 4.8 – An example where the first evaluated stage is a metagenerator. Left: main with intermediate and final versions; Right: Integrated metaprograms from original and intermediate main versions.	75
Figure 4.9 – Emulation of the traditional top-down inside-out stage evaluation order in the integrated model; delayed stage evaluation is forced with quasi-quotes and inlining.	79
Figure 5.1 – Reviewing the compilation sources in Sparrow: Project manager view (<i>left</i>), initial main source (<i>middle</i>), assembled stage source (<i>bottom</i>), final main source after staging (<i>middle right</i>).	92
Figure 5.2 – Storing the source code of all stage metaprograms and their outputs (main program transformations).	93
Figure 5.3 – Build system and compiler interaction sequence diagram regarding metaprograms.....	95
Figure 5.4 – Control flow for the staging-aware build process; starting process is ‘build’ (top-left).	97
Figure 5.5 – Precise error reporting for compilation stages using the chain of generated sources.	99

Figure 5.6 – Instrumenting #line directive for better error reporting in C#: The compilation error occurs in the generated file but is reported in the original file.....	101
Figure 5.7 – A compile-time debugging session in Sparrow; highlighted items 1-9 are discussed in text.	103
Figure 5.8 – Interaction between the compiler and integrated development environment for supporting compile-time source-level stage debugging.....	104
Figure 5.9 – Extracting line mappings for a compilation stage: The assembled stage AST (top), the original source and the compilation stage source (bottom right) and the line mappings generated by each AST node (next to each of the AST nodes, referring to elements of the bottom left table).	106
Figure 5.10 – Alternative views for inspecting AST values in Sparrow: an expression tree view (left), a viewer unparsing AST to source code (left, top) and a graphical tree view (left, bottom).	108
Figure 5.11 – <i>Left</i> : source-level debugging support for runtime stages in C#, Java and Delta regarding the different availability options of stage source text and its respective debug information; <i>Right</i> : the split responsibility in using source text and debug information for debugging sessions.	112
Figure 5.12 – Example in C# illustrating the instrumentation of the generated code with debugger break instructions to stop execution in the context of the generated code during a debug session.	113
Figure 5.13 – Sample implementation of a multi-stage language with runtime metaprogramming relying on: (i) staging extensions preprocessing; and (ii) runtime code generation, loading and invocation via the language reflection facilities.	115
Figure 6.1 – The two alternative contexts for aspect weaving.	118
Figure 6.2 – Compile-time staging and aspect weaving options.	119
Figure 6.3 – Runtime staging (compiled language case) and aspect weaving options.	120

Figure 6.4 – Runtime staging (interpreted language case) and aspect weaving options.	120
Figure 6.5 – Source-level aspect weaving as a batch of AST transformation programs.	128
Figure 6.6 – Complete overview of all aspect transformation batches occurring during compilation.	129
Figure 6.7 – Interaction sequence diagram between the build system, compiler and aspect weaver.	134
Figure 6.8 – Sample workspace showing source files generated by staging and aspect transformations.	136
Figure 6.9 – Tracing compile errors in the source transformation pipeline involving staging and aspects; source names refer to the workspace of Figure 6.8.	138
Figure 6.10 – Full-scale source-level debugging of aspect programs when they are actually applied during compilation.....	139
Figure 7.1 – The recursively-repeated structure of exception handling patterns generated via metaprogramming; <i>stmts</i> may contain exception handling code following the main pattern.	164
Figure 7.2 – Deploying library metaprograms to generate exception handling patterns.	164
Figure 7.3 – Modular composition of exception-handling patterns as decorator stacks.	172
Figure 7.4 – Architecture of generative model-driven tools: (1) interactive model editing; (2) code generation from models; and (3) tags inserted in the generated source code to carry model information and enable model reconstruction.....	175
Figure 7.5 – Common growth of application code around the originally generated code; future custom extensions and updates eventually lead to bidirectional dependencies.....	176

Figure 7.6 – The primary maintenance issues in the deployment of generative model-driven tools either individually (left) or collectively (right).	177
Figure 7.7 – The refined model-driven process with an inverted responsibility through staging: programmers deploy generator macros to insert generated code on-demand and in-place without affecting the originally produced ASTs by the MDE tools. The second stage applies translation on compile-time staging, or evaluation (translation and execution) on runtime staging.	178
Figure 7.8 – Applying the generative MDE process with runtime staging; the application composes intermediate or source text and deploys the language reflection API for compilation and invocation (JIL stands for Java Intermediate Language, CIL for the Common Intermediate Language of .NET).	179
Figure 7.9 – Extending the staged model-driven generator proposition to offer self MDE deployment.	181
Figure 7.10 – Examples of generated interfaces: <i>Left</i> : Original application GUI authored by the interface builder; <i>Middle</i> : Custom toolbar authored as a separate interface; <i>Right</i> : Composing the two previous interfaces through AST manipulation.	183
Figure 7.11 – <i>Top-left</i> : Ecore model of the target class hierarchy; <i>Top-right</i> : Code structure (AST) generated by the model; <i>Bottom</i> : Deployment code for loading and converting the model to AST, performing manual updates through AST editing and inlining the final AST code.	184

List of Tables

Table 3.1 – Comparison of languages regarding the requirements for integrated metaprogramming. The symbol ↓ means that the feature is offered by the language with certain limitations (more details in the language-specific discussion sections). .	52
Table 4.1 – Code generation examples for quasi-quotes and escapes, showing that they do not involve staging.	67
Table 6.1 – Ability to implement aspects under different categories of multi-stage languages and for the different possible weaving contexts and subjects.....	121
Table 6.2 – Ability to implement aspects under different categories of multi-stage languages and for the different possible weaving contexts and subjects.....	126
Table 6.3 – Overview of the basic elements offered by our AOP library	132
Table 7.1 – Comparison of AOP and metaprogramming in the context of exception handling.....	174

“The computer programmer is a creator of universes for which he alone is the lawgiver. No playwright, no stage director, no emperor, however powerful, has ever exercised such absolute authority to arrange a stage or a field of battle and to command such unswervingly dutiful actors or troops.”

- Joseph Weizenbaum

Chapter 1

Introduction

“I'd rather write programs to write programs than write programs.”

- Dick Sites

The essence of programming is the transformation of the algorithmic logic required to solve a certain problem into a program able to produce the solution. This way, we express the logic only once and then use the program for any future occurrences of the particular problem. Let's consider the trivial example of deciding whether or not a number is prime. It is clear that even for relatively small numbers this becomes a tedious and error-prone task. Yet, the algorithmic logic required for the solution is pretty straightforward and can be easily turned into a program that yields the correct results.

The same line of thinking can be applied at a higher level, considering the problem at hand to be the transformation of an algorithm into a program. In this sense, we do not just want to deploy the logic of a particular problem into a program that solves it, but rather have this logic turned into a higher-order program able to generate particular problem solutions. For example consider the various design principles and patterns available. When programming we take them into account and try to incorporate them into our code when applicable. However, if we are to use a design pattern in two different contexts we usually end up implementing it twice. Clearly, it would be more efficient to transform our knowledge regarding the pattern and its use into a single algorithm that will then be deployed for each target context to provide a full pattern implementation.

The process described above refers to the creation of programs that generate or transform other programs, a method known in general as *metaprogramming*. Metaprogramming can help achieve various benefits [Sheard01], the most typical of which is performance. It provides a mechanism for writing general purpose programs without suffering any overhead due to generality; rather than writing a general

purpose but inefficient program, one writes a program generator that generates an efficient solution from a specification. Additionally, by using partial evaluation it is possible to identify and perform many computations at compile time based on a-priori information about some of the program's input, thus minimizing the runtime overhead. Another significant metaprogramming application is the reasoning about object-programs. It is possible to analyze and discover properties of the object-program that can be used to improve performance, provide assurance about the behavior of the object program, or provide object program validation. Examples of reasoning metaprograms include flow analyzers and type checkers. Finally, metaprogramming can be used for code reuse at a macroscopic scale. Currently, languages support code reuse through functions, generics, polymorphism, classes and interfaces. However, there are recurring code patterns that cannot be abstracted and reused with the above approaches. Since metaprogramming transforms or generates code operating on code segments, it is possible to capture and abstract the recurring code using some structured representation and deliver it as a directly reusable unit.

In general, metaprogramming involves a normal program p and a metaprogram mp that when deployed will produce a transformed program p' or create a new program p'' based on it (Figure 1.1). The language in which the original program p is written is called the *object language* while the language in which the metaprogram mp is written is called the *metalanguage*. If the object language and the metalanguage are the same, it is a case of *homogeneous* metaprogramming, while if they are different it is a case of *heterogeneous* metaprogramming. In any case, the abstract view of metaprogramming process illustrated in Figure 1.1 has multiple incarnations matching the different forms that the involved items p , mp , p' and p'' may take.

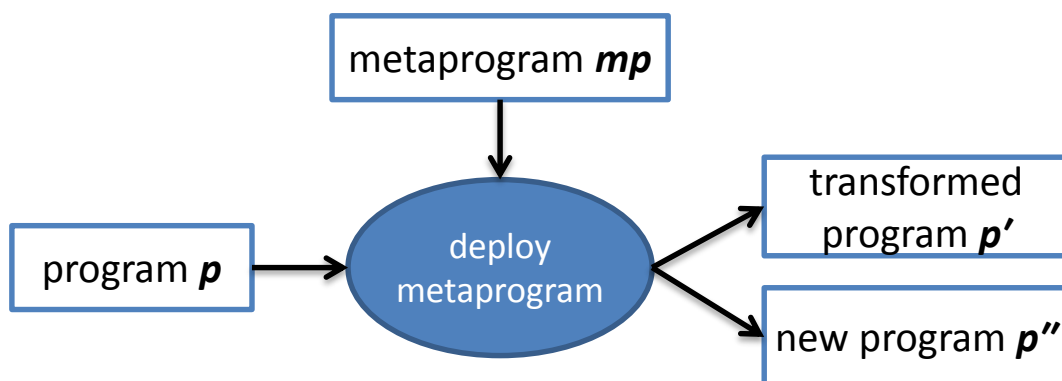


Figure 1.1 – Abstract view of the metaprogramming process

One incarnation that is probably the first encounter many programmers have with the notion of metaprogramming is the *C Preprocessor* (CPP) [Kernighan]. CPP receives C source code fragments as input (p) and generates other C source code fragments as output (p'), with its metaprogram logic (mp) being specified as a text-substitution macro system denoted with the `#define` directives. While part of the C language, CPP is often implemented as a separate program enabling its usage in different contexts, where the input and output programs are not necessarily C code fragments. In fact, CPP, as a text-based system, is unaware of the syntax and semantics of any language or program that deploys it, dictating only the syntax used for the metaprogramming logic, i.e. the macro definitions.

Other incarnations may involve generating a new program in an entirely different language than the one used for the original program. The lexical analyser generator *Lex* and the parser generator *Yacc* are tools deploying such a form of metaprogramming. *Lex* uses a pattern matching language to describe lexical elements of a target language and *Yacc* uses a context free grammar to specify its syntactic structure, while both generate C code to perform the lexical and syntax analysis for the target language. This is a typical example of using metaprogramming for improved performance; rather than writing a general purpose but inefficient program for lexical or syntax analysis, one writes a program generator that generates an efficient solution from a specification [Sheard01].

There are also scenarios where the original, the final program as well as the metaprogram are all specified in the same language. A language exemplifying this metaprogramming scenario is Lisp [McCarthy][Steele]. In Lisp, the textual representation of a program is simply a human-readable description of the same internal data structures (linked lists, symbols, number, characters, etc.) as would be used by the underlying Lisp system. Lisp macros operate on these code structures and Lisp code has the same structure as lists so macros can be built with any of the list-processing functions in the language. In this sense, any operation that Lisp performs on a data structure, Lisp macros can perform on code. It is important to note that code in Lisp is represented in syntactic forms called *s-expressions* [McCarthy]. Compared to the text-based representation of the CPP, this is far more expressive as it constitutes a structured representation that allows the metaprogram logic to inspect internals of a

code fragment, modify it at a syntactic level and algorithmically combine it with other code fragments.

Another similar scenario concerns program staging in *Multi-Stage Languages* [Sheard00][Taha97][Taha04]. Multi-stage languages allow programmers to explicitly state the evaluation order of the various computations specified in a program with each stage of evaluation essentially generating code for future stages or the main program. As such, each stage can be seen as a metaprogram that receives as input staged source code fragments and produces as output other source code fragments that are inserted in the main program being evaluated. The source code fragments used in this process are specified in some structured syntactic form, usually an Abstract Syntax Tree (AST), thus allowing the metaprogram logic to easily iterate over the represented source code and manipulate its contents.

Metaprogramming for transforming the source code of a program is not limited to affecting specific parts of the program, as is the case with macros or staged computation, but it can also affect the entire program. *Aspect-Oriented Programming* (AOP) [Kiczales97] is a programming paradigm that can be considered to follow such a metaprogramming approach. AOP models crosscutting functionality that can be introduced at specific locations of a target program. In this sense, the metaprogram input is the entire program code, the metaprogram logic consists of the specification of aspects (typically performed in a separate language) while the output of the metaprogramming process is the target program code combined with the crosscutting functionality introduced by the aspect. In this case, the input and output programs are typically in the same form that can be either source code, some intermediate code or AST representation, or even binary code.

Finally, the input or output of a metaprogramming process may not even be in code form. For example, consider *Model-Driven Engineering* (MDE) [Kent], a software development methodology utilizing domain models as primary engineering artifacts. In MDE, a model can be transformed to another model (model-to-model transformation) or a source code implementation in a target language (model-to-source transformation) while existing source code can be used to extract a model (source-to-model transformation). Such transformations can be seen as metaprograms

written in some language (e.g. OCL [OMG12]) that operate on some input form (source, or model) and produce output in another form (again either source or model).

Despite current efforts to effectively support metaprogramming, there are still open issues ranging from aspects of language design to integrated development environment (IDE) facilities and practices for metaprogram deployment. Within this thesis, we explore the field of metaprogramming in general and the domain of multi-stage languages in particular and focus on facilitating the practicing of metaprogramming, effectively paving the way for its adoption as a large-scale development discipline.

We continue detailing the motivation for our work and elaborate on issues identified towards facilitating metaprogram development. Finally, we present our contributions in the field, reflecting the software engineering propositions, deployment practices and implementation efforts required for addressing the identified issues.

1.1 Motivation

Many languages provide some support for metaprogramming and the amount of meta-code being developed has started to grow rapidly over the past few years. However, metaprogramming is still being treated as a special feature that is separated from the main language. Metaprograms are usually developed and deployed with no resemblance to normal programs. From a developer perspective, they tend to lack common notions like files and modules, while from a deployment perspective they typically adopt a macro invocation policy with no state sharing or the notion of a main control flow. Moreover, metaprogramming lacks effective support for project management, editing automations and source-level debugging, something restricting larger-scale metaprogram developments. There seems to be no particular intention for such lack of features other than the inherent implementation complexity when trying to accommodate them in languages and tools. As metaprograms are programs, it is irrational to offer diverse development styles amongst the two worlds and to actually provide fewer facilities to metaprograms. In this direction, we emphasize the necessity for a *methodological integration* between metaprogramming and normal programming, featuring common software practices and development tools.

In the same sense, we consider that certain principles or paradigms traditionally found in normal programming could be directly adopted or extended to apply in a metaprogramming context. A representative paradigm in this respect is AOP that currently supports only normal programs. Since metaprograms are full-scale programs, they may involve cross-cutting concerns of their own, thus also requiring AOP support. Effectively, this means that current AOP practices should be refined to take into account the potential deployment on metaprograms. Additionally, there is the opposite direction of aspects requiring metaprogramming support [Zook]. In this context, and considering the previous discussion about AOP as a metaprogramming method that transforms the original program with cross-cutting functionality, it is interesting to explore the potential of deploying aspects as transformation programs expressed in the same language. Such a notion would directly enable metaprogramming support for aspects while also benefiting from any facilities offered by the existing metaprogramming system due to language sharing.

Another promising direction is the potential of deploying metaprogramming towards reuse. Traditional language features for reuse like functions, classes, modules, etc. may not always suffice to capture and express arbitrary recurring code patterns of any scale. For instance, consider *Design Patterns* [Gamma] that constitute directives for solving common software engineering problems. There is no outcome that can be directly reused as a program fragment; rather a description of how to solve the problem in different situations, meaning that they should be manually adapted and applied for each instance. Metaprogramming can achieve a higher level of reuse by abstracting over code fragments and allowing the direct reuse of implemented code templates that are instantiated through custom design parameters. A similar example relates to the creation of reusable exception handling structures. In real-life software systems, normal code and exception handling code is frequently tightly coupled and specified within syntactically distinct blocks disallowing the adoption of traditional language reuse approaches like inheritance, abstraction, polymorphism and genericity towards modular and directly reusable exception handling code. With metaprogramming, source code becomes a first-class value allowing syntactic structures like exception handlers to be parameterized as reusable and directly deployable units that can be inserted on demand in a target program.

Finally, still in the context of metaprogram deployment, we believe that metaprogramming has the potential to overcome maintainability issues involved in source code automation tools. For instance, consider Model-Driven Engineering [Kent][Schmidt] where a source code skeleton is generated based on some model and is then manually extended to produce the complete application. The manual extensions cannot be easily reconciled with the original model while any model updates cannot be directly incorporated in the code base as regenerating the source code skeleton will discard the manual extensions. Through the use of metaprogramming, we can overcome such issues by encapsulating application generators as metaprograms. In a metaprogramming context, a model or the source code it generates need not be external resources viewed separately from the code; in fact they can constitute metaprogram data that can be used along with custom deployment logic to freely mix model code and manual code extensions as part of a metaprogram.

1.2 Contributions

This work targets the field of metaprogramming focusing on compiled languages and explores the various aspects of the metalanguage design and its features as well as the tools needed to provide the desired metaprogramming support. We strongly believe that *metaprogramming is essentially programming* and we want to support it with joint techniques and tools rather than treat it like a special feature. Only through proper language features and tool support can metaprogramming become a development approach usable in large-scale applications. Our ultimate goal is twofold. On the one hand we want to provide a methodology for the development of *Integrated Metaprogramming Systems* covering aspects of the design process, the compilation and runtime execution, the system architecture and component interoperation as well as the supporting tools. On the other hand we want to derive a code of practice that will utilize metaprogramming techniques to achieve reusability by supporting aspect-oriented programming, implementing design pattern generators and exception handling templates, and facilitating the automation of source code generation in the context of Model-Driven Engineering. Overall, the contributions of this thesis are the following:

- We identify a set of requirements related to language features, software engineering, and programming environments in order to support integrated metaprogramming.
- We introduce the notion of *integrated compile-time metaprograms* and propose a multi-stage metaprogramming model that realizes stages as independent coherent programs assembled from specific meta-code fragments present in the source code.
- We provide an *integrated tool chain* that supports metaprograms with tools and features similar to those used for normal programs. In particular, our system offers: (i) integration of metaprograms and generated programs in the workspace manager facilitating source browsing and editing features; (ii) a build system aware of the staging process delivering typical build and deployment tools for metaprograms; (iii) meaningful compile-error reporting in the context of metaprogramming; and (iv) full-scale source-level debugging of metaprograms and generated programs.
- We propose a methodology for introducing aspect-oriented programming in the entire staging pipeline and support aspect deployment as AST transformations expressed in the same language.
- We develop a practice that achieves reusable implemented design patterns by utilizing metaprograms as pattern generators.
- We propose an approach for implementing reusable exception handling patterns with compile-time metaprogramming.
- We address the maintenance issues of model-driven code generation by refining the engineering process to encapsulating application generators as metaprograms.

The work in this thesis has been implemented in the context of the untyped object-based language *Delta* [Savidis05], [Savidis10]. As such we do not focus on type checking issues or type system properties. Nevertheless, our propositions are orthogonal to typing and can be well applied to any language, either typed or untyped, as long as they offer the required support for metaprogramming. A significant reason for choosing Delta in particular, was that we had access to both language and IDE source code so as to implement the proposed metaprogramming extensions. Another reason was the architectural split between the Delta language components, i.e.

compiler, virtual machine and debugger that enabled implementing these metaprogramming extensions in an organized and modular fashion.

1.3 Outline

This thesis is organized as follows. Chapter 2 provides some background information on metaprogramming and discusses related work, focusing on language and tool support for metaprogramming. Chapter 3 presents the key requirements identified for integrating metaprogramming and normal programming and reviews existing metalanguages against these requirements. Chapter 4 introduces the integrated metaprogramming model and elaborates on aspects of metalanguage design and implementation methods. Additionally, it compares the expressiveness of the proposed model against the prevalent existing model and presents selected case studies that evaluate and demonstrate the software engineering value of our proposal. Chapter 5 focuses on tool support and details the extensions required to programming environment facilities to accommodate metaprogramming. Chapter 6 explores the adoption of aspect-oriented practices in the context of metaprogramming along two orthogonal directions: (i) offering aspect support in the entire staging pipeline; and (ii) realizing aspects as batches of transformation programs without requiring dedicated languages. Chapter 7 discusses the deployment of metaprogramming towards advanced software practices including design pattern generators, exception handling templates, and staged model-driven generators. Finally, Chapter 8 summarizes the key points of this thesis, draws key conclusions and discusses directions for future research.

Chapter 2

Related Work

“The greatest part of a writer's time is spent in reading, in order to write: a man will turn over half a library to make one book.”

- Samuel Johnson

2.1 Background Information

Metaprogramming involves generating, combining and transforming source code, so it is essential to provide a convenient way for expressing and manipulating source code fragments. Expressing source code directly as text is impractical for code traversal and manipulation. Alternatively, intermediate or even target code representations are very low-level to be deployed. Currently, ASTs are widely adopted for source code representation, due to their ease of use and because they retain the original code structure.

Although ASTs provide an effective method for manipulating source code fragments, manually creating ASTs for source fragments usually requires a large amount of statements making it hard to identify the actually represented source code [Weise]. Thus, ways to directly convert source text to ASTs and easily compose ASTs into more comprehensive source fragments were required. Both requirements have been addressed by existing languages through a feature known as *quasi-quotation* or *quasi-quoting* [Bawden]. Normal quotes skip any evaluation, thus interpreting the original text as code. Quasi-quotes works on top of that, but instead of specifying the exact code structure, they essentially provide a source code template that can be filled with other code. To better illustrate this notion we briefly discuss its support in various staged languages with a simple example.

Consider the following Lisp macro which generates the multiplication of the argument X by itself. Definitions after the *backquote* operator ``` are not directly evaluated but are interpreted as a code fragment value (i.e. an AST). The reverse of

backquote is the *unquote* operator ```, which causes evaluation of such a code fragment value (sort of lazy evaluation). The result is the expression `(* 5 5)` yielding 25.

```
(defmacro square (X)
  `(* ,X ,X))
(square 5) ; 25
```

The same example in *MetaML* [Sheard98] follows, where surrounding *brackets* `<...>` are used to turn code fragments to ASTs (called delayed computations in MetaML) and *escape* `~` enables combination of such ASTs within bracket expressions. This means that *square* (see below) contains the AST of `5*5`. Finally, *run* is used to directly evaluate an AST (called execute of delayed computation in MetaML) which in our example evaluates to 25.

```
val code = <5>;
val square = <~code * ~code>;
val result = run square; (* 25 *)
```

In *Converge* [Tratt05] the example looks quite similar, with a few syntactic changes regarding the staging annotations: code within quasi-quotes `[|...|]` is converted to AST, while *insertion* `${...}` and *splice* `$<...>` operators relate to *escape* and *run* of MetaML.

```
code := [| 5 |]
square := [| ${code} * ${code} |]
result := $<square> // 25
```

Finally, the same example in *Metalua* [Fleutot07a] follows. Quasi-quotes are denoted with `+{...}` while `-{...}` implies splicing if inside quasi-quotes or execution otherwise.

```
result = -{
  block:
    code = +{ 5 }
    return +{ -{code} * -{code} }
} -- 25
```

2.2 Language Support for Metaprogramming

2.2.1 Macro Systems

Macro systems operate on a source file by specifying certain input sequences that should be mapped to output sequences according to some user defined procedure. Macro systems may be language agnostic operating solely on input text and using some fixed syntax to define the mapping procedure. Such systems are usually called

external preprocessors as they are typically used externally with respect to the language translator. On the other hand, they may be a built-in language mechanism being aware of the language syntax and semantics and may even use the full language itself to specify the transformation logic, with the Lisp macro system being a typical example of this category. Figure 2.1 shows the processing diagram of a macro system.

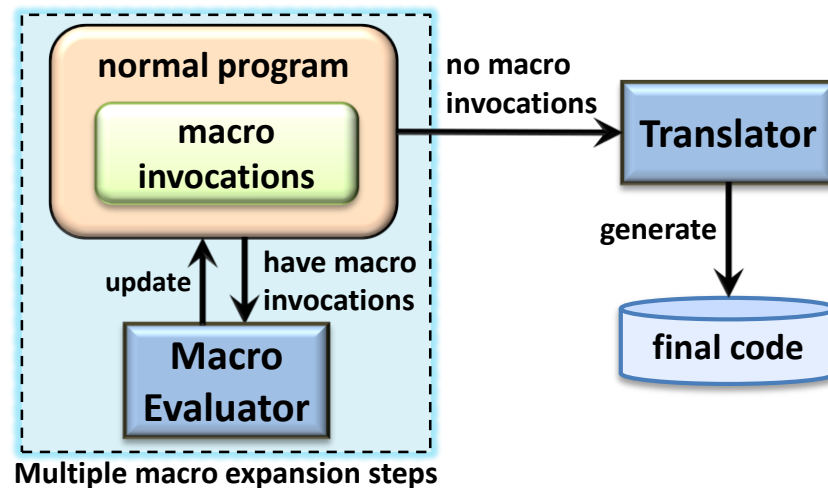


Figure 2.1 – Processing diagram for macro systems

2.2.1.1 Text-Based

One of the most common categories of text-based macro systems is that of external preprocessors. They are external tools that can be used independently from the main programming language and they actually have no knowledge of the target language. They process the language source file as simple text and perform the necessary text substitutions producing the final source file that will later be given to the language compiler or interpreter. As a result, a programmer using an external preprocessor should be extra cautious to avoid common pitfalls like wrong operator precedence or duplicate side effects.

One popular external preprocessor is the *C preprocessor (CPP)* used in the C programming language [Kernighan]. In many C implementations, *CPP* is a separate program invoked by the compiler as the first part of translation. The preprocessor handles directives for source file inclusion, macro definitions and conditional computation. Specifically for the macro definition it supports object-like and function-like macros, token concatenation and token stringification, features

especially helpful for performing compile time computations and writing code generating code. For example, a macro whose parameters represent partial source-code units, types or names may be used to generate code at compile time.

```
#define GEN_ATTRIBUTE(id, type) \
    void Set_##id (const type& val) { id = val; } \
    const type& Get_##id (void) const { return id; }
class Point {
private:
    double x, y;
public:
    GEN_ATTRIBUTE(x, double)
    GEN_ATTRIBUTE(y, double)
}
```

One relatively unknown technique that utilizes the CPP to generate repeating code structures for similar operations executed on a list of items is the *X Macro* [Bright]. The technique involves a macro definition enumerating the list items and passing them as arguments to the supplied X macro that will operate on the list items. For example, consider a list of colors for which we want to automatically generate code for enumerated values, string names, RGB values, etc. To achieve this we can use the following code:

```
#define COLORS(X) \
    X(red, "red", 255, 0, 0), \
    X(green, "green", 0, 255, 0), \
    X(blue, "blue", 0, 0, 255)

#define ID(id, name, r, g, b) id
enum Color { COLORS(ID) };

#define NAME(id, name, r, g, b) name
char *ColorNames[] = { COLORS(NAME) };

#define RGB(id, name, r, g, b) {r, g, b}
int RGBValues[][3] = { COLORS(RGB) };
```

Another general purpose macro processor is the *m4* [Turner]. In contrast to the *CPP*, *m4* supports a freeform syntax, rather than line based syntax as well as a high degree of macro expansion (arguments get expanded during scan and again during interpolation). It also provides file inclusion, text replacement, parameter substitution conditionals and loops. Most importantly though, *m4* supports macros that can generate other macros, as shown in the example below. This feature makes it more expressive compared to *CPP*, always with respect to metaprogramming support.

```
define(`definedefinex',`define(`definex',`define(`X',`xxx')')')
definex X # -> definex X
```

```

definedefineX X      # ->  X
defineX X            # ->  xxx

```

Overall, even though macro processors are widely used and provide some basic metaprogramming support, they tend to be insufficient for full scale metaprogramming. The main reason is that code is treated as text and therefore we cannot inspect the internals of code supplied as an argument. This way, we cannot inject additional code at specific point of an input source code unit or even modify them at a syntactic level. Intuitively one would like to have some sort of AST representation to manipulate code either for iteration purposes (read) or for editing (write).

2.2.1.2 Syntax-Based

A typical example of a language that provides a macro system which has the full language itself available for the transformation logic is Lisp. A fundamental distinction between Lisp and other languages is that in Lisp, the textual representation of a program is simply a human-readable description of the same internal data structures (linked lists, symbols, number, characters, etc.) as would be used by the underlying Lisp system. Lisp macros operate on these code structures and Lisp code has the same structure as lists so macros can be built with any of the list-processing functions in the language. In this sense, any operation that Lisp performs on a data structure, Lisp macros can perform on code. The programmer specifies a macro definition stating its name and arguments as well as the code replacement. The macro definition is specified using *defmacro* keyword and the code replacement may include the special backquote, unquote and splicing operators to allow representing code structures that may have arguments injected in them both in evaluated and unevaluated forms. Any special syntax appears only in the macro definition; the macro invocation resembles a normal function call.

Another language that provides a syntax-based macro system is *Scheme* [Dybvig09]. Scheme macros operate on ASTs and allow making sophisticated decisions based on a node's context within the tree. They are introduced using the *define-syntax* keyword followed by associations of new syntactic keywords with transformation procedures created using *syntax-rules* or *syntax-case* clauses and a simple pattern matching sublanguage. Scheme macros are hygienic and respect the scoping rules of the rest of

the language. This is assured by special naming and scoping rules for macro expansion and avoids common programming errors that can occur in the macro systems of other programming languages. Again, macro invocations bear a close resemblance to procedures (both are indeed *s-expressions*) but they are treated differently. The compiler first checks an expression for symbols defined as syntactic keywords in the current lexical scope and tries to expand the macro treating the items in the tail of the expression as arguments without compiling code to evaluate them and this is performed recursively until no macro invocations remain.

MS² [Weise] is a macro system for infix syntax languages like C. It is programmable in a minimal extension of C offering a template substitution mechanism based on Lisp's quasi-quotes and uses a type system to guarantee at macro definition time that all macros and macro functions only produce syntactically valid program fragments. There is no support for hygiene, but instead requires programmer intervention to avoid variable capture errors. From an implementation perspective, code template operators make the language context sensitive thus involving changes in the parser; the parser should perform type analysis in order to parse macro definitions or parse user code that invokes macros.

Dylan [Bachrach99] also provides a macro system based on skeleton syntax tree (SST) approach and using a set of rewrite rules. Initially the program is parsed using a "phrase" grammar able to understand only tokens and balanced delimiters. Then the SST is traversed parsing the built-in forms and looking for macros to expand. When a macro is encountered, the tokens that constitute the macro body are compared against the set of rewrite rules and when the appropriate rule is matched the arguments are accordingly substituted with the pattern matched values and the output of the macro is parsed again until no macros are found. However Dylan's syntax is not significantly more flexible than LISP's, and its macro related syntax is heavyweight, as it is a separate language from Dylan itself. *JSE* [Bachrach01] is a macro system for Java following a similar approach to Dylan macros. It differs from Dylan as exploits Java's compilation model to offer a full procedural macro system instead of one relying only on rewrite-rules. This also allows the pattern matching and rewrite rule engine to be less complex since standard Java control and iteration constructs can be used along with it. Finally, JSE can package and reuse syntax expansion utilities in the same way

as any other Java code, while the elements of its pattern matching engine are open to programmer extension.

Marco [Lee] is an expressive and safe macro system macro that is independent from the target language. It is based on the observation that the macro system need not know all the syntactic and semantic rules of the target language but need only enforce specific rules (syntax and name bindings for free and captured variables) that can be checked by special oracles utilizing unmodified target-language compilers and interpreters. These oracles are deployed by submitting specially crafted programs to the target-language processor and then analyzing any resulting error messages. Macro provides static types, conditionals, loops, and functions, making it Turing-complete, while supporting target language fragments as first-class values through a quasi-quote like syntax. For safety, it uses macro-language types to check target-language syntax, and uses dataflow analysis to check target-language naming discipline. However, for any language to be supported the programmer has to provide the appropriate language-specific oracles, while to guarantee safety it requires the target language to produce descriptive error messages that identify locations and causes of errors.

[Burmako] introduces a variety of *macro flavors* for supporting compile-time metaprogramming in *Scala* [Odersky]. In particular apart from the *def macros* (typical Lisp-style macros), it supports *dynamic macros*, *string interpolation macros*, *implicit macros*, *type macros* and *macro annotations*. Each flavor encompasses a different way that macros are presented and can be used by users, supporting various applications scenarios like language virtualization, type providers, materialization of type class instances, type-level programming, external domain-specific languages and language extensibility.

2.2.2 Multi-Stage Languages

Multi-stage languages extend the multi-level language [Glück95][Glück96] notion of dividing a program into levels of evaluation and make them accessible to the programmer through special syntax called *staging annotations* [Taha97]. Such annotations are introduced to explicitly specify the evaluation order of the program computations, effectively generating code segments for future stages (Figure 2.2).

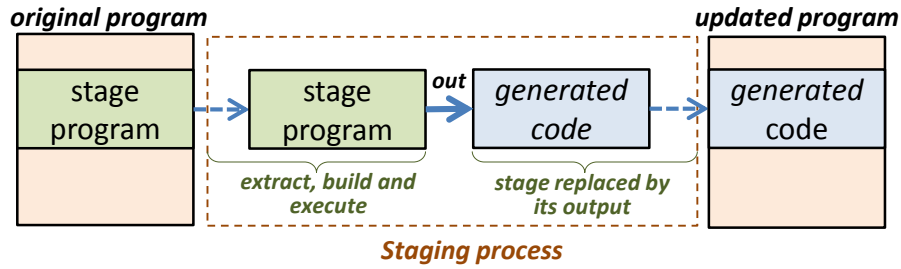


Figure 2.2 – General staging process in multi-stage languages.

In this sense, a staged program is a conventional program that has been extended with the appropriate staging annotations. Below we detail the basic staging annotations adopting the syntax of MetaML, an extension to the functional programming language ML, as it constitutes a good basis for general-purpose multi-stage programming (other multi-stage languages discussed later have similar annotations both in syntax and semantics).

- *Brackets* ($\langle_ \rangle$) construct a code fragment delaying its computation
- *Escape* ($\sim_$) combines code fragments (i.e. already delayed computations)
- *Run* ($\text{run } _$) executes a code fragment that corresponds to a delayed computation
- *Lift* ($\text{lift } _$) constructs a code fragment from a ground value, such that the code fragment represents the ground value

Brackets can be inserted around any expression to delay its execution. For example:

```
-| val result0 = 1+5;
val result0 = 6 : int
-| val code0 = <1+5>;
val code0 = <1+5> : <int>.
```

In a typed language like MetaML, the *brackets* of a delayed computation are also reflected in the type. The type in the last declaration is $\langle \text{int} \rangle$, read “Code of Int”. The code type constructor is the primary device that the type system uses for distinguishing delayed values from other values and prevents the user from accidentally attempting unsafe operations such as $1 + \langle 5 \rangle$.

Additionally, MetaML prevents accidental collisions of variables introduced within brackets with program variables of the same name regardless of the context in which a code fragments is executed; in other words brackets respect hygiene.

Escape allows the combination of smaller delayed values to construct larger ones. This combination is achieved by splicing the argument of the *escape* in the context of the surrounding *brackets*:

```
-| val code1 = <(~code0, ~code0)>;
val code1 = <(1+5, 1%+5)> : <int * int>.
```

Escape combines delayed computations efficiently in the sense that the combination of the subcomponents of the new computation is performed while the new computation is being constructed, rather than while it is being executed. This subtle distinction is crucial for staging can make a big difference in the run-time performance of the delayed computation.

Run allows the execution of a code fragment. It is a very important construct since it is the only way to execute code fragments and therefore achieve the multi-stage computations. The use of *run* in MetaML can be illustrated with the following simple example:

```
-| val result = run <1+5>;
val result = 6 : int.
```

Lift allows us to inject values of ground type into a value of type code. Both *brackets* and *lift* construct code, but *lift* does not delay its argument; it first evaluates it and then constructs a representation for its value:

```
-| val code3 = lift 1+5;
val code3 = <6> : <int>.
```

Lift is restricted only to ground types and is not available for functions, as there is no general way of computing a source-level representation for a function. It is not a fundamental staging construct, since MetaML allows variables that are bound at one level to be used at a higher level. Nevertheless, *lift* helps producing code that is easier to understand, because constants become explicit.

Another multi-stage extension of ML is *MacroML* [Ganz]. MacroML supports inlining, recursive macros and the definition of new binding constructs and views macros as multi-stage computations. This eliminates the need for freshness conditions and tests on variable names, and provides a compositional interpretation that can serve as a basis for designing a sound type system for languages supporting macros.

Multi-stage programming has also been explored in the context of compiled languages. For example, *MetaOCaml* [Calcagno01] is a metaprogramming extension of OCaml and is essentially a compiled dialect of MetaML. In fact, it is implemented through a combination of ASTs, gensym and runtime reflection [Calcagno03]. It differs from MetaML in that it safely handles ML's side-effecting.

Multi-stage languages are typically homogeneous; nevertheless [Eckhardt] introduces the notion of implicitly heterogeneous multi-stage programming, where object language and metalanguage are different but the details of the representation are handled by the metalanguage designer one and for all, allowing the programmer use a familiar interface to execute generated code, thus maintaining a homogeneous multi-stage programming experience. This enables existing generators to target different languages without requiring any changes, and maintains the type correctness guarantees of generated code as long as the translation itself is type preserving. Work on heterogeneous metaprogramming also includes F# [Syme], an ML variant for .NET, whose quasi-quoted values are generated into .NET code thus enabling interoperability of code fragments across various .NET languages.

Most multi-stage languages focus on code generation and its optimization, but they cannot operate as code analyzers as offering functionality to destruct or traverse quoted expressions may cause the static type-safety guarantees to be violated. [Viera] proposes a multi-stage language with intentional analysis that relaxes the static safety in favor of flexibility and offers a homogeneous meta-system that allows observing the structure of its object programs. The latter is achieved by a pattern matching mechanism that is used to inspect the structure of quoted expressions and destruct them into their component subparts.

There is also work towards applying multi-stage programming in the context of imperative languages. For example, *Metaphor* [Neverov04][Neverov06] is a C# [Hejlsberg] based language for expressing multi-stage programs in a strongly-typed, imperative, object-oriented environment. It provides static type-checking of later stage code and offers a type reflection capability to discover information about types at run-time. Metaphor allows this reflection system to be incorporated into the language's staging constructs, thus allowing the generation of code based on the

structure of types. Additionally, it treats both types and code as first-class values, covering all four kinds of program reflection: code and type generation and analysis.

Combining multi-stage programming with imperative features is difficult, mainly due to scope extrusion, in which free variables can inadvertently be moved outside the scopes of their binders. In this direction, [Kameyama08][Kameyama09] introduced a two-level language that provides delimited control operators (shift and reset), while assuring statically that all generated code is well-formed. The language was based on a two-level calculus with control effects and a sound type system, that was later extended by [Kokaji] consider polymorphism. The key idea to prevent scope extrusion was to restrict control effects to the scope of generated binders, that is, to treat generated binders as control delimiters. Effectively, this means that code in quasi-quotes should not have observable side effects. This requirement was later relaxed in *Mint* [Westbrook], a multi-stage extension of Java [Arnold]. In *Mint*, specific terms can be declared as *weakly separable* meaning that they do not have observable side effects that involve code values. With escaped terms being weakly separable, *Mint* can guarantee that no code value (and hence no future-stage variable) can leave the scope in which it is generated. This way, type safety is retained, while the system becomes more expressive; for example, in *Mint* it is possible to throw an exception in a code generator, or accumulate code in a for-loop. However, there are still restrictions; for instance restricting non-local operations within escapes to final classes practically excludes a significant part of the standard Java library. Recently, [Rhiger] took this one step further by introducing a type system that supports multiple stages, evaluation under future-stage binders, as well as open code manipulation.

2.2.3 Runtime Metaprogramming

In traditional multi-stage languages like MetaML and MetaOCaml, code generation occurs during program execution, so the term runtime metaprogramming – RTMP – has been associated with multi-stage languages. However, runtime metaprogramming is not limited to multi-stage languages. It can also be achieved through *reflection*, a language facility allowing examining or modifying the structure and behavior of program code during execution. This is typically supported by providing the compiler and loader as libraries that can be used at runtime. This way, a programmer may compose dynamic text containing source code and use the provided API to compile,

load and deploy the dynamic code. For example, Figure 2.3 shows how two mainstream languages, C# and Java, can offer metaprogramming through their reflection API. For C# in particular, another option is to use the forthcoming *Roslyn* technology [Ng].

C#
<pre>string source = "class Test { public void func() { System.Console.WriteLine(\"Hello World\"); } }"; CodeDomProvider provider = CodeDomProvider.CreateProvider("CSharp"); CompilerParameters cp = new CompilerParameters(); cp.GenerateInMemory = true; CompilerResults result = provider.CompileAssemblyFromSource(cp, source); // invoke the compiler Assembly assembly = result.CompiledAssembly; // get the compiled assembly Type type = assembly.GetType("Test"); // get generated class information Object o = Activator.CreateInstance(type); // create an instance of the generated class type.GetMethod("func").Invoke(o, null); // get and invoke 'func' method, printing Hello World</pre>
Java
<pre>String source = "public class Test { public void func() { System.out.println(\"Hello World!\"); } }"; JavaCompiler compiler = ToolProvider.getSystemJavaCompiler(); StringSourceJavaObject src = new StringSourceJavaObject("Test", source); Iterable<? extends SimpleJavaFileObject> fileObjects = Arrays.asList(src); compiler.getTask(null, null, null, null, null, fileObjects).call(); // invoke the compiler Class<?> clazz = ClassLoader.getSystemClassLoader().loadClass("Test"); // load generated class Object o = clazz.newInstance(); // create an instance of the generated class clazz.getMethod("func").invoke(o); // get and invoke 'func' method, printing Hello World</pre>

Figure 2.3 – Runtime code generation and execution through reflection: C# (top) and Java (bottom).

Lightweight Modular Staging (LMS) [Rompf] is a library-based approach for runtime multi-stage programming in Scala. It tries to avoid quasi-quote syntax and instead uses only types to distinguish between binding times. Essentially, while multi-stage programming provides staging support for all language constructs by default but requires the programmer to explicitly annotate staged code, LMS involves no explicit staging but requires operations on staged types to be explicitly provided by the programmer as traits.

'C [Engler], *Jumbo* [Kamin] and *DynJava* [Oiwa] are all two-level languages that support metaprogramming through dynamic code generation. They extend their respective base languages (C for 'C and Java for Jumbo and DynJava) with quasi-quote operators that allow programmers specify code fragments in the original source language level, thus facilitating compositional code generation. Jumbo performs type checking when code is generated at runtime, requiring only the end result to be correct; this yields better expressiveness but no safety guaranties. On the contrary, 'C and DynJava impose restrictions to what can be expressed but offer static typing facilities that perform such checks during compilation. From the two, 'C is more expressive but also provides less safety guaranties, as code fragments lack context

information and thus inconsistencies may arise in generated code. To offer stronger type safety guaranties, DynJava relies on annotating dynamic code fragments with the type and context information they involve. Such annotations, referred to as *code specifications*, are essentially typed quasi-quotes that contain meta-data about the free variables they contain. This information can then be used to check inconsistencies at compilation time. Similar type safety guarantees in a dynamic code generation context are offered by the *Mnemonics* [Rudolph] library that spots most byte-code verification errors at compile time of the generator. However, Mnemonics specifies generated code using directly byte-code instead of quasi-quoted source code.

Finally, in interpreted language implementations (e.g. Lisp or Scheme interpreters) there are no separate compilation and execution steps, only a single interpretation step. This way, code generation (e.g. by a macro invocation) is not separated by the execution of normal program code. In this sense, we also include such language cases in the runtime metaprogramming family.

2.2.4 Compile-Time Metaprogramming

Program staging methods may also be applied during program compilation. In this context, compile-time staging, known also as compile-time metaprogramming - CTMP, supports the evaluation of staging definitions that transform the main program during the compilation phase. Examples of languages supporting such a compilation scheme include Template Haskell, Converge and Metalua.

Template Haskell [Sheard02] is a two-stage language that provides metaprogramming facilities through quasi-quotes and splicing. Quasi-quotes `[| ... |]` can be inserted around ordinary Haskell concrete syntax fragments (analogous to *brackets*) to represent Haskell programs ([Mainland] proposes an extensible quasi-quotation mechanism for Haskell providing access to many object languages), and the splice operator `$` that accepts an argument of type expression (analogous to *escape*) can force the evaluation of some code within quasi-quotes during their construction. The splice operator can also be used outside the quotations with the meaning of evaluating its argument at compile time (analogous to *run*). In case the splice occurs at top level, its argument may also have a declaration type in order to introduce new data types, classes or instance declarations. There is also the lift operator that transforms a value to an expression type (analogous to *lift*). Template Haskell also allows the

programmer to query the state of the compiler’s internal symbols, called reification. This essentially provides a general way to get compile-time information about declarations allowing the programmer to write *reifyDecl* *f* and get a data structure that represents the value declaration for *f*. Finally, it is important to note that quasi-quotes respect lexical scoping in the sense that every occurrence of a variable is bound to the value that is lexically in scope at the occurrence site in the original source program, before any template expansion, while they also respect macro hygiene.

Converge [Tratt05] follows the compile-time metaprogramming approach of Template Haskell deploying it in a dynamically typed object-oriented language. It has the same annotations semantics but with minor lexical differences; the operator to perform compile time evaluation, called *splicing operator*, is denoted as $\$<...>$ and the evaluation of an expression within a quasi-quote that copies the resulting AST into the AST being generated by the quasi-quote is called *insertion* and denoted as $\$\{...\}$. Converge also provides a lift operator and respects macro hygiene.

Metalua [Fleutot07a] is an extension of *Lua* [Ierusalimschy] that offers compile-time metaprogramming support. It has similar staging annotations but a significantly different underlying philosophy [Fleutot07b], strictly separating compile-time metaprogramming meta-levels. Metalua introduces the concept of moving between layers of meta-levels using the annotations $+ \{...\}$ (equivalent to *brackets*) and $- \{...\}$ (equivalent to *run*, unless it is nested inside a $+ \{...\}$ when it is equivalent to *escape*) operators. The code executed at compile time is referred to as ‘level 0’, and the result of the compilation as ‘level 1’. Of course there can be other levels as well; for instance if the compile-time metacode itself relies on generation, it will be produced by code that is executed in level -1.

Compile-time metaprogramming is not limited to program staging methods; there is a wide range of systems supporting compile-time metaprogramming through macros, templates, Meta Object Protocols (MOPs) [Kiczales91], traits, compile-time reflection, compiler functionality reification, etc.

C++ templates [Stroustrup] constitute a Turing Complete [Veldhuizen03] functional language interpreted at compile time [Abrahams][Veldhuizen96] as part of the language type system that can be exploited to perform compile-time computations.

Essentially, C++ can be seen as a two-stage language where the first stage consists of the interpretation of the templates (denoted by the `< >` tags used in both declarations and instantiations) and the second stage the compilation of the non-template code. For example, consider the following template definitions used to calculate the Fibonacci sequence. All types are resolved during compilation so all instantiations of the `Fibonacci` struct, along with their enclosed enumerated field value are determined before code generation. Thus, when generating code for the expression `Fibonacci<5>::value` the result will be the constant value 8.

```
template<int n> struct Fibonacci
{ enum { value = Fibonacci<n-1>::value + Fibonacci<n-2>::value }; };
template<> struct Fibonacci<0> { enum { value = 1 }; };
template<> struct Fibonacci<1> { enum { value = 1 }; };
printf("%d", Fibonacci<5>::value); //8, calculated at compile-time
```

In addition to the template system, the *C++11* standard [Becker] adds an extra metaprogramming approach through const expressions; the keyword *constexpr* can be used on functions that meet some requirements, allowing them to be invoked during compilation if their arguments are constants.

Nemerle [Skalski04] is a statically-typed class-based language compiled to CIL (.NET binary) supporting compile-time metaprogramming through its macro system. It uses quasi-quotes denoted as `<[...]>` to express the syntax tree of the enclosing expression and a splice operator `$` to force an evaluation during the quasi-quote construction. *Nemerle* macros are defined explicitly with the *macro* keyword so there is no need for additional syntax in their invocation. They are invoked like a normal function, but instead of generating a run-time function call the macro is executed at compile time and the generated code is inlined for further processing. *Nemerle* macro invocations can take place within the body of a function, providing a simple compile-time function call, but they can also be used at various other places in the source file targeting some specific declaration (class, field, method, property, event or argument). This is achieved by supplying meta-attribute properties for the macros at their definition and essentially stating where the macro will be used and the compilation phase at which it should be processed. Based on these attributes, the macro will take some specific arguments (for instance the class or the parameter being targeted by the macro) that are automatically supplied by the compiler. This seems a rather intrusive approach and it also requires knowledge of the compilation stages. Nevertheless,

automatically supplying parameters based on the context of the macro use, introduces the interesting idea of context aware macros.

OpenC++ [Chiba] and *OpenJava* [Tatsubori], are extensions of C++ and Java respectively offering compile-time MOPs. In this approach, metaobjects (meta-class instances) are available during compilation, providing a compile-time reflection mechanism that is used to manipulate source code and provide class translation through a method called *type-driven translation*. Essentially, after parsing the original source, the system generates a class metaobject for each defined class. Then it deploys the class metaobjects to translate the target class to normal language syntax (if further metaobjects are involved in the generated code the same process continues recursively until we have normal language syntax) and finally sends it to the original language compiler for normal compilation. In this sense, both systems operate as separate source-to-source preprocessors. The main difference between *OpenC++* and *OpenJava* is that the former utilizes ASTs as the data structure for source code manipulation, while the latter focuses on a data structure that represents the logical structure of an object-oriented program. Another system with a compile-time MOP is *Jasper* [Nizhegorodov]; however it focuses on syntactic extensions rather than on code generation.

The *Jakarta Tool Set* (JTS) [Batory] provides a set of domain-independent tools for creating domain specific languages. JTS consists of two tools: *Jak* and *Bali*. *Jak* is a metaprogramming extension of Java supporting AST constructors to create typed quotations AST manipulation through a tree walk and hygienic generation facilities. *Bali* is a parser generator for creating syntactic extensions based on a BNF grammar with regular-expression repetitions. A JTS component consists of a *Bali* grammar file for the extension syntax and a set of *Jak* files for the extension semantics. JTS is related with the compile-time MOPS with its elements having direct counterparts: *Jak* corresponds to the metalanguage, while *Bali* corresponds to MOP itself. In this sense, JTS represents arbitrary syntactic extensions as GenVoca components like MOPs represent class-specific extensions as meta-classes.

SafeGen [Huang05] is a metaprogramming tool for generating Java programs. It features *cursors* that are variables ranging over all entities satisfying a first-order logic formula over the input program, and *generators* which use cursors to output code

fragments. Generators are written in a quasi-quotation style, giving the system a great deal of flexibility, while their safety is statically determined by constructing first-order logic sentences and checking their validity through a theorem prover. Since validating first-order logic sentences is undecidable and the theorem prover may not terminate for certain queries, SafeGen poses a time limit and maintains soundness by providing warnings on the time-terminated queries.

Genoupe [Draheim] is a C# extension that supports defining and applying program generators at compile-time. It is based on parameterizing classes over types and values, and offers a `@foreach` keyword to loop over fields or methods of a type parameter and generate code for each match. *Genoupe* offers a type system that offers a high degree of static safety; however it cannot guarantee that generated code is always well-typed as the deployed parameters (e.g. types) do not carry enough constraints to allow such checking. Also, it can only generate new programming elements, not add functionality to existing ones.

Another C# extension for compile-time metaprogramming is *CTR* [Fähndrich]. *CTR* offers compile-time reflection and utilizes a high-level construct called *transform* to write code for inspection and generation in a pattern matching and template style, avoiding at the same time the complexities of reflection APIs. It avoids the explicit quoting and unquoting conventions to keep new syntactic constructs to a minimum and relies on meta-variables and a few keywords. It also provides the benefits of staged compilation as well-formedness of generated code is statically checked. The latter applies also for compiled transform entities, meaning they can be distributed normally, while maintaining their safety guarantees. *CTR* offers better safety guarantees than *Genoupe* and enables extensions by combining patterns and generators within a single transform.

MorphJ [Huang08][Huang11] is another language supporting pattern-based reflective declarations. It improves expressiveness through nested patterns that elaborate the outer-most pattern with blocking or enabling conditions without sacrificing safety. In particular, *MorphJ* refines the type system of *CTR* with a modular type system and offers a both high level and safer solution as it can separately type-check generic classes and catch errors early. Additionally, it does not require introducing concepts

outside of the base language (as the *generator* and *transform* of CTR); instead code generation is incorporated with the concept of generic classes.

Meta-trait Java [Reppy] is a compile-time framework that allows both generation and introspection of code focusing on member-level patterns. It introduces user-customizable traits that are parameterized over types, values, and names offering compile-time pattern-based reflection. Traits support a uniform, expressive and type-safe way for metaprogramming without resorting to AST manipulation, with their type system incorporating a hybrid of structural and nominal subtyping. *PTFJ* [Miao] generalizes the trait functions of Meta-trait Java, combining pattern-based reflection with traits and providing language features for manipulating sets of member declarations like giving them names, manipulating their domains using set operations, and passing them as arguments to traits.

Mython [Riehl] is a variant of the Python programming language that supports extending the compilation process through an extended quotation mechanism. Apart from the code being quoted, Mython quotations accept an additional parameter that is used to both parse the quoted code and extend the compile-time environment (unlike other user code, the quotation parameter is evaluated at compile-time). This approach allows embedding other languages by specifying compile-time definitions able to translate the embedded code into Python. Such translations return host language abstract syntax, and the possibly modified compile-time environment. Programmers can bind new names in the compile-time environment by using a special translation function provided by the compiler. Finally, compiler built-in functions become first class values enabling the programmer customize their functionality and thus offering support for domain-specific optimizations.

MetaFJig [Servetto10][Servetto13] is a Java-like language where class definitions are first class values that can be built on existing classes through a set of primitive composition operators, namely *sum*, *restrict*, *alias*, and *redirect*. Compilation is based on a series of meta-reduction steps called *compile-time execution* that try to derive non constant class declarations in the context of the current metaprogram. This process is guaranteed to be sound by interleaving meta-reduction steps with type checking that dynamically detects class composition errors. The latter allows for a modular approach enabling compile-time execution to be defined on top of type-

checking and execution system of the underlying language. MetaFJig also ensures *meta-level soundness*; this means that no typing errors during compile-time execution can originate from meta-code that has already been compiled (e.g. library code). This is not granted for other approaches like C++ templates.

Backstage Java (BSJ) [Palmer] is a Java extension for compile-time metaprogramming, supporting algorithmic, contextually-aware generation and transformation of code. It features the following properties: (i) non-local changes are effected without incurring confusing side-effects; (ii) execution order is dependency-driven to retain determinism in case of non-local changes; and (iii) conflicts between independent metaprograms are automatically detected. To achieve these, BSJ uses a novel difference-based metaprogramming approach that regards metaprograms not as simple program transformations but as transformation generators. In particular, ASTs record any changes made on them using *edit scripts* and each metaprogram is executed on a different AST copy without observing changes from other metaprograms it does not depend on. Eventually, the generated edit scripts are merged to produce the final program. Any failure in the merge operation corresponds to a metaprogram conflict and is reported appropriately by the system.

Fan [Hongbo] is a compile-time metaprogramming system for OCaml that features a unified abstract syntax representation defined using polymorphic variants and supports nested quasi-quotes for the full language syntax, allowing them to be overloaded and customized by the programmer. It also provides support for syntactic extensions based on *delimited, domain-specific languages* (DDSLs) that can be implemented as libraries. In fact, the quasi-quotation mechanism is just another DDSL library that is bundled with the compiler.

2.2.5 Modifiable Syntax and Semantics

Programming languages usually have a predefined fixed syntax that is a core part of their specification. Some of them have a uniform and flexible syntax (i.e. Lisp) that allows them to specify new syntax constructs into the language while others (for instance Camlp4 [Rauglaudre]) allow extending some fixed grammar entries. The motivation behind such syntax extensions is to allow programmers extend the language and customize it according to their needs. Especially in languages that support macro systems they are extremely useful, since they allow conveniently

incorporating popular syntactic entities from other languages as well as embedding *domain specific languages* (DSLs) in the main language [Czarnecki].

Lisp has traditionally been the language that used its macro system to add syntactic extensions to the language core and indeed, elements of the standard Lisp syntax are implemented as macro extensions. It has such a minimalistic syntax that essentially everything written in it extends the basic language. The programmer uses the *defmacro* keyword as well as the *quote*, *backquote*, *unquote* and *splicing* operators to express the macro name and its parameters along with the code that will be replaced at each occurrence of the macro invocation. A simple example of a Lisp macro adding a square construct to the language is the following:

```
(defmacro square (X)
  '(let ((Temp ,X))
      (* Temp Temp)))
```

Scheme also supports introducing new syntactic constructs to the language through its macro system. New syntactic extensions are defined by associating keywords with transformation procedures, or *transformers*. Syntactic extensions are defined globally using top-level *define-syntax* forms or within the scope of particular expressions using *let-syntax*, *letrec-syntax*, internal *define-syntax*, or *fluid-let-syntax* while transformers are created with *syntax-rules*, *syntax-case*, or some implementation-dependent mechanism. Syntactic extensions are expanded into core forms at the start of evaluation (before compilation or interpretation) by a syntax *expander*. The expander is invoked once for each top-level form in a program. If the expander encounters a syntactic extension, it invokes the associated transformer to expand the syntactic extension, and then repeats the expansion process for the form returned by the transformer. If the expander encounters a core syntactic form, it recursively processes the sub-forms, if any, and reconstructs the form from the expanded sub-forms. Information about identifier bindings is maintained during expansion to enforce lexical scoping for variables and keywords. Below, we have an example of a Scheme macro implementing a *foreach* construct based on the *map* function.

```
(define-syntax foreach
  (syntax-rules ()
    ((foreach element in list body ...)
      (map (lambda (element)
              body ...)
            list))))
```

Dylan also provides syntax extensions based on its skeleton syntax tree (SST) approach. Initially the program is parsed using a “phrase” grammar able to understand only tokens and balanced delimiters. Then the SST is traversed parsing the built-in forms and looking for macros to expand. When a macro is encountered, the tokens that constitute the macro body are compared against a set of rewrite rules and when the appropriate rule is matched the arguments are accordingly substituted with the pattern matched values and the output of the macro is parsed again until no macros are found. A sample Dylan extension is illustrated below.

```
define macro when
  { when (?test:expression) ?body:body end }
  => { if (?test) ?body end if }
end macro;
```

Metalua is able to dynamically extend its syntax based on its approach to separate meta-levels. The programmer may introduce syntax extensions regarding prefix, infix and suffix expression modifiers or new statements based on dedicated keywords, and does that by simply dropping one meta-level and plugging-in the desired extensions to the Metalua parser by specifying the necessary lexical (keywords), syntactic (token sequence) and semantic (function to perform the extension logic) information. All code that follows in the file in the original meta-level will be parsed with these extensions enabled. Code in lower meta-levels is not affected making it clear when and where a syntax change takes effect and preventing syntax-changing code from interfering with itself. Metalua extensions are easy to use but there are limitations in what can be parsed the programmer needs to have a very good understanding of the underlying parser and compiler. The extensions can be directly incorporated in the standard language but composition of multiple and possibly advanced extensions can be challenging or even impossible. An example of a Metalua syntactic extension that adds a power operator to the language is available below.

```
-{ block:
  mlp.lexer:add{ "let", "in" }
  mlp.expr:add{ "let", mlp.id, "=", mlp.expr, "in", mlp.expr,
    builder = let_in_builder }
  local function let_in_builder (x)
    local variable, value, expr = unpack (x)
    return +{
      function (-{variable})
        return -{expr}
      end (-{value}) }
    end
}
```

Converge supports syntactic extensions by introducing the concept of a DSL block on top of the standard splice operator. This allows arbitrary blocks of text to be embedded in a Converge file that will be treated as complete, localised DSLs embedded into the language. When the Converge tokenizer encounters a DSL block, the text on the next level of indentation is left unparsed and is passed as a raw string to a user defined DSL implementation function that is called at compile-time to parse the text and return an AST. To this end, Converge provides a number of convenience functions to capture standard idioms of DSL parsing as well as DSL AST creation. Converge limits the ways in which new syntax can be embedded into the language, but allows any syntax to be embedded, without interfering with the main language. This allows a clean separation between, and composition of, languages even when DSLs are embedded within each other. However this also means that DSLs can have relatively limited interaction with each other. Here is an example of a Converge DSL block.

```
func timetable(dsl_block, src_infos):
    parse_tree := CEI::dsl_parse(dsl_block, src_infos, \
        ["Premium", "Cheap"], [], GRAMMAR, "start")
    return Translator.new().generate(parse_tree)
$<timetable>:
8:25 "Exeter St. Davids" Premium
10:20 "Salisbury" Premium, Cheap
11:49 "London Waterloo"
```

Nemerle also has built-in support for syntax extensions but it is limited to a number of fixed places of the language grammar. There are two forms of syntax extensions. Using the first one, the programmer is able to add new parsing rules that will be triggered by a set of user defined keywords and operators. When the parser encounters one of them at a valid position it executes a special parsing function for syntax extension related to the corresponding token. The second one allows specifying that a given part of program input will not be interpreted by the main parser, but will be passed to a function that will perform user defined parsing based on the stream of tokens. This stream consists of grouped and matched opening and closing brackets ({}, (), [] and <[]>) that they call token groups. In the definition of the syntax extension macro a parameter is annotated to accept such a token group and will be given the nearest group of tokens from the input during its invocation. An example of a macro defining a syntax extension in Nemerle is the following:

```
macro xml_literal (tokens : Token)
  syntax ("xml", tokens) { // process 'tokens' }
def x = xml <person><name>John</name></person>;
```

While this approach allows embedding an arbitrary syntax into the language, it is limited by the fact that token groups have certain separators (comma for () and [], semicolon for {} and <[]>) that have a special meaning and therefore cannot be used as part of the embedded syntax. This means that the syntax extensions allowed by the language are restricted to those conforming lexically to Nemerle and having the same token tree structure.

SugarJ [Erdweg] is a Java based language built on the grammar formalism *SDF* [Heering] and the transformation system *Stratego/XT* [Bravenboer], introducing the notion of *sugar libraries* as an approach for syntactically extending a programming language within the language. A sugar library is like an ordinary library, but can, in addition, export syntactic sugar for using the library. Each piece of syntactic sugar defines some extended syntax and a transformation – called *desugaring* – of the extended syntax into the syntax of the host language. For example, the following code shows the definition of a sugar library for pairs along with an example usage.

```
//library source
package pair;
public class Pair<A,B> { ...pair implementation as a generic class... }

package pair;
import org.sugarj.languages.Java;
import concretesyntax.Java;
public sugar Sugar {
  context-free syntax
    "(" JavaType "," JavaType ")" -> JavaType {cons("PType")}
    "(" JavaExpr "," JavaExpr ")" -> JavaExpr {cons("PEExpr")}
  desugarings
    desugar-pair-type
    desugar-pair-expr
  rules
    desugar-pair-type:
      PType(t1, t2) -> [| pair.Pair<~t1, ~t2> |]
    Desugar-pair-expr:
      PEExpr(e1, e2) -> [| pair.Pair.create(~e1, ~e2) |]
}

//application source
import pair.Sugar;
public class Test {
  private (String, Integer) p = ("12", 34);
}
```

Sugar libraries maintain the composability and scoping properties of ordinary libraries and thus are good candidates for embedding DSLs into a host language. As libraries they can also be applied on other libraries, effectively supporting syntactic extensions in the definition of other sugar libraries. Finally, then can be imported across meta-levels to activate language extensions in user programs or act on all meta-levels uniformly to enable syntactic extensions in metaprograms.

In this thesis, we focus on a unified language for both normal programming and normal programming. On the contrary, syntactic extensions typically distinguish between programming elements available in the metalanguage or the normal language. In this sense, the support for syntactic extensions deviates from our main direction and thus it is not considered as a requirement in the metalanguage design.

2.3 Tool Support for Metaprogramming

2.3.1 Error Reporting

Having proper error reporting for compilation errors is essential in the context of metaprogramming as the erroneous code may be generated from other code and never appear in the original source. However, most languages that support metaprogramming provide very limited error reporting for compilation errors originating from generated code. Typically, the error is reported directly at the generated code with no further information about its origin or the context of its occurrence. Below we examine some of the few cases that offer a more sophisticated error reporting mechanism for compilation errors.

C++ compilers (e.g. *Microsoft Visual Studio Debugger*, *GDB*) provide fairly descriptive messages regarding compilation errors occurring within template instantiations. Using these messages provided, the programmer may follow the instantiation chain that begins with the code of the initial instantiation that caused the error (typically user code) and ends with the code of the instantiation that actually triggered the error (probably library code). Essentially, these error messages represent the execution stack of the template interpreter. While potentially informative and able to provide accurate information to experienced programmers, template error messages are quite cryptic for average programmers and require significant effort to locate the actual error. Unfortunately, this is the common case for nontrivial meta-programs and

applies especially to libraries with multiple template instantiations (e.g. *Boost* [Abrahams]).

Converge provides some error reporting facilities related to meta-programming by keeping the original source, line and column information for quoted-code and retaining it at splice locations (injections into the program AST). For runtime errors, this approach works fine but is limited by the single source code location that can be associated with a given virtual machine instruction, not allowing for a complete trace of the error. For compile-time errors, *Converge* can track down the source information of the quasi-quotes and associated insertions (i.e. any AST creation) to provide a detailed message. However, it fails to provide information about the splice locations, which actually involve staging execution. This means that any error originating in generated code cannot be properly traced back to the code that actually produced it. Finally, any compile error reported is presented only with respect to the original source, thus providing no actual context regarding the temporary module (i.e. computation stage) being executed to perform the splice.

[Hirzel] presents a macro system that deploys contractual checks to offer better compile-time error reporting. Each macro is associated with a pre-condition and a post-condition that are checked during macro expansion. If an error occurs, it reports the violated contract appropriately, blaming the macro call for pre-condition violations or the macro definition for post-condition violations. This approach avoids errors that are hard to understand, because they refer to implementation details of the macro. However, contracts for complex macros may be difficult to express, while they require a lot of effort on behalf of the programmer.

2.3.2 Debugging

Many metalanguages are extensions of existing languages with metaprogramming features. For example, MetaOCaml is a multi-stage extension of the OCaml language, MetaML is a higher-order extension of the ML language for staged metaprogramming, Template Haskell is a Haskell extension that adds compile-time metaprogramming facilities and Metalua extends Lua with a complete macro system. The base languages typically provide some debugging facilities (e.g. *ocamldebug* for OCaml, *MLWorks* debugger for ML, *GHCi* debugger for Haskell and *Ldb* for Lua); however their counterparts that support metaprogramming fail to provide similar

debugging tools. This is not limited to metalanguage extensions alone; there is a general lack of debugging support for most languages offering metaprogramming facilities. For instance, in Template Haskell the closest a programmer gets to tracing and debugging the code being executed at compile time is using a compiler flag to show the expansion of the top-level code splices as they happen. Similarly, the only debugging facility in Metalua and Converge is the pretty printing of AST values (for example Converge’s `CEI::pp_itree` library function). Overall, debugging functionality in such languages is mainly limited to the primitive “printf debugging”, being far from adequate when dealing with complex metaprograms. We continue with some exceptions of languages and tools that offer more advanced debugging support in the context of metaprogramming.

C++ support for metaprogramming is based on its template system that is essentially a functional language interpreted at compile time. The *C++11* standard also introduced functions executed at compiled-time when declared with the keyword *constexpr*. There are C++ debuggers (e.g. *Microsoft Visual Studio Debugger*, *GDB*) that allow source level debugging of templates, but only in the sense of tracing the execution of the template instantiation code and matching it to the source containing the template definition. However, during compilation there is neither a way to debug template interpretations nor is there support for tracing functions declared as *constexpr*. A step towards the former is *Templight* [Porkolab], a debugging framework that uses code instrumentation to produce warning messages during compilation and provide a trace of the template instantiation. Nevertheless, it is an external debugging framework not integrated into any development environment and relies on the compiler generating enough information when it meets the instrumented code. Finally, there is no programmer intervention; the system provides tracing but not interactive debugging.

D [Alexandrescu] is a statically typed multi-paradigm language that supports metaprogramming by combining templates, compile time function execution, and string mixins. *Descent* [Descent], an *Eclipse* plugin for D code, provides a limited compile-time debugging facility for simple templates and compile-time functions. However, the debugging process does not involve the normal execution engine of the language; instead it relies on a custom language interpreter for both execution and debugging functionality.

Nemerle and its IDE, *Nemerle Studio*, provide support for debugging macro invocations during compile time. *Nemerle* macros are actually compiler plug-ins that have to be implemented in separate files and modules and are loaded during the compilation of any other file that invokes them. Since they are dynamically linked libraries with executable code, it is possible to debug them by debugging the compiler itself; when a macro is invoked, the code corresponding to its body is executed and can be typically debugged. However, the development model posed, requiring macros to be separated, is restrictive and the macro debugging process is rather cumbersome.

DrRacket (formerly *DrScheme*) [Findler], a Scheme IDE, provides a facility for debugging macro code. Specifically it has a macro stepper that allows the programmer to see all macro transformations step by step. The macro stepper has a good interface that highlights and matches macro arguments and their usage while also providing information about their source location and properties. Furthermore, it is possible to view the variable renaming steps used for the macro hygiene. The macro stepper is a significant debugging aid for metaprogramming especially in a macro-extensible language where extensions can be stacked in a “language tower”.

Lisp IDEs like *AllegroCL* [Franz] and *LispWorks* [LispWorks] provide good macro debugging facilities. For example, both provide tracers able to show any function or macro invocation, their arguments, environments as well as their results. Additionally, they provide macro steppers for inspecting evaluations step by step. During each step, macro invocations can either be directly evaluated to provide their result or they can be expanded to show the resulting code and then stepped further into for a detailed trace. A variety of stepping options are provided, (step to through call, step to call, step to value, next step, step to end and step to cursor) enabling the programmer perform a very targeted trace and thus significantly minimizing the time and effort needed to debug complex macros.

JSE provides a macro expand facility that can generate the result of a macro expansion given an input string containing the macro call. The macro call can either be fully expanded or expanded one level at a time, thus enabling smart editors to selectively macro expand marked regions of program source. JSE also offers macro tracing flags for output regarding pattern matching and the binding of pattern

variables. Still, the debugging functionality is far from what is typically supported by a Java debugger.

2.3.3 IntelliSense

Source editing support, also known as *IntelliSense* [Microsoft], is a feature of great importance as it provides a significant aid for the programmer and speeds up the software development process. Most IDEs provide such support generally in the form of tooltips that display information regarding various language expressions and symbols (e.g. argument names, symbol types, external documentation, etc) or in the form of auto-completion. Unfortunately, to our knowledge, there are few IDEs that support source editing facilities with respect to the metaprogramming features of their targeted language.

As previously discussed, C++'s metaprogramming is based on its templates. Visual Studio for C++ provides Intellisense information for all template code. There is auto-completion support for templates functions and template classes and their members as well as tooltips that appear when hovering over a symbol or typing a template instantiation or template function call that provide information about the template and its parameters. It should be noted of course, that C++ templates cannot produce any code other than their instantiations with specific types. This makes proving of source editing features quite easier as all information required for the source editing features is available directly from the source files.

Nemerle also provides support for source editing with respect to its macros. Macros are compiler plugin declared in separate files and modules and handled as special class definitions. This way, it is easy to provide auto-completion support for their invocations. More importantly though, when hovering over a macro invocation Nemerle provides a tooltip with information about the macro (point of definition and possibly keywords associated with it) and its result, i.e. the code that will be produced by the invocation and injected into the file being compiled. This way, programmers can directly see what the macro code they type translates to and have a better overview of the resulting source. On the downside, while typing a macro invocation there is neither Intellisense support for its arguments nor a prompt to show its correct syntax in case it is a syntactic extension macro.

Leksah [Nicklisch], an IDE for Haskell that provides some support for Template Haskell as well, provides source editing that supports metaprogramming. More specifically, it provides auto-completion that takes into account symbols that are brought into scope through metacode (i.e. top level splices). This is indeed very helpful, as the programmer can acquire and use information that is not directly available when reviewing the source. A drawback though is that the auto-completion information is not automatically updated during editing but requires the source file to be compiled first.

2.3.4 Browsing

Another important feature provided by many popular IDEs is source browsing. It allows programmers to have a better overview of their code structure providing them with easy access and navigation across modules, classes, functions, variables, etc. As previously discussed, in a metaprogramming system a significant part of the final executable code is introduced using metacode, and is therefore not easy (or in some cases even possible) to browse through it. It is therefore evident that source browsing should be an essential feature for integrated metaprogramming systems. Nevertheless, we have observed that only a few existing systems provide such a feature.

There are IDEs for C++ (e.g. *Microsoft Visual Studio* and *SunStudio*) that provide source browsing features that support the language's metaprogramming constructs, i.e. templates. For example, Visual Studio provides a project based class view as well as a file based scope view that provide information about classes, namespaces, functions, macros, constants and type definitions. Everything related to a template is properly marked and associated with its template arguments. However, the note about C++ templates not able to produce any code other than their instantiations with specific types applies here too; all relevant source browsing information is already available on the original source so one can extract them with a manner similar to normal source browsing.

Open Dylan, a native code compiled Dylan implementation, has a full-feature IDE that among others provides source browsing facilities. Specifically, it provides an overview of the various definitions (macros, classes, constants, functions, variables, etc) present within a module. Most importantly though, it shows all macro invocations that introduce any additional definitions along with the definition introduced

themselves. This is really important as it directly shows the programmer all definitions available in the normal code. A small drawback though is that this information is not automatically refreshed during editing; a compilation of the source file is first required to generate the browsing information.

A similar feature can also be found in Leksah. In Template Haskell, top level splices may introduce declarations and Leksah provides a source browser that displays information about such generated declarations. Again though, we have the same drawback; the information is only refreshed after a compilation of the file containing the top level splice.

All in all, there is limited source browsing support both in quantity and in quality. Many more IDEs could provide source browsing systems and the existing ones could provide additional information or generally a better overview of the metacode and its outcome. Especially for multi-stage languages, there is no such support. However, each compilation stage may have a different set of available definitions that may even not be available in the original source, so it especially valuable having a good overview of what's available at each stage.

Chapter 3

Requirements

“The formulation of a problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill.”

- Albert Einstein

The practicing of metaprogramming is strongly affected by the ability of related language features and tools to effectively support software engineering. In this context, we identify some prominent software engineering requirements that are essential for the integrated practicing of metaprograms and normal programs and review existing metalanguages and systems against the identified requirements.

3.1 Analysis

We define and elaborate a set of requirements derived from the weaknesses of existing metalanguages that compromise the software engineering of metaprograms. That is, they are criteria directly affecting the practicing of metaprogramming and constitute an important aspect of our work. Overall, we consider metaprogramming to be an art fundamentally harder to normal programming. However, the restricted software engineering support by existing languages makes it even harder, sometimes rendering metaprogramming to a dark art for average programmers. As it becomes evident it is the bar regarding metaprogramming facilities that is actually raised by such requirements to a level similar to normal programming.

3.1.1 Exploiting Normal Language Features and Tools

One of the most important requirements towards integrating normal programs and metaprograms is the full exploitation of all normal language features and tools in the context of the metalanguage. It is essential that metaprogrammers experience the metalanguage as an extension on top of the normal language, rather than as a restriction of it. For instance, if the normal language supports classes, threads and modules, the metalanguage should also provide them in the same manner. While such

a requirement may seem apparent at a first glance, as we discuss under the related work, it is not currently met by the available compile-time metaprogramming languages. We continue by briefly explaining the implications of this requirement in the practicing of compile-time metaprogramming.

There are two reasons justifying why the exploitation of all normal language constructs in the metalanguage is critical. Firstly, in implementing a metaprogram one should not be given fewer features than what is offered in implementing a normal program. Secondly, even when alternative equivalent facilities are offered, it is difficult and painful for programmers familiar with the normal language to learn and deploy a different set of constructs for similar programming tasks. Overall, the normal language should be fully reused in expressing and organizing the metaprogramming logic, with additional syntax and semantics introduced only where necessary.

To outline the importance of this requirement, consider C++ templates [Stroustrup], which can support some level of compile-time metaprogramming. As previously discussed, templates reflect a special-purpose functional language, interpreted during compilation, being fundamentally different from the class-based imperative low-level nature of the normal language. The latter requires so radically diverse approaches to cope with similar computation problems that reuse of design or code is disabled. For instance, consider the following normal C++ code for the Fibonacci sequence:

```
int fibonacci(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
printf("%d", fibonacci(5));    //8, calculated at runtime
```

To perform the same computation during compile-time with templates requires a feature known as recursive template specialization:

```
template<int n> struct Fibonacci
{ enum { value = Fibonacci<n-1>::value + Fibonacci<n-2>::value }; };
template<> struct Fibonacci<0> { enum { value = 1 }; };
template<> struct Fibonacci<1> { enum { value = 1 }; };
printf("%d", Fibonacci<5>::value);    //8, calculated at compile-time
```

Clearly, the two versions are fundamentally different. In particular, the second one is far less readable and obvious as it widely deviates from the common style of the language and involves custom coding practices.

In general, we argue that there should be no particular language-oriented distinction between normal functions and metafunctions. They may differ in terms of their operational role, with the latter typically implementing some AST manipulation logic, but other than that there should be no fundamental difference between them.

Apart from the thorough exploitation of all language features, the entire set of language tools should be reusable as well, including the compiler, runtime library, virtual machine and debugger. In other words, metalanguage development should avoid reinventing the wheel and emphasize tool reuse, while appropriately extending or refining where needed according to the extra metaprogramming requirements. In particular, the original compiler and virtual machine may be extended to translate and execute respectively both normal programs and metaprograms. Similarly, by extending the original debugging system of the language, source-level debugging of metaprograms should be facilitated as with normal programs.

3.1.2 Supporting Context-Free and Context-Sensitive Generation

Metaprogramming is commonly used like a macro system to transform a source fragment by inserting extra source code through code generation directives. In this framework, there are two possible options (Figure 3.1) in controlling the insertion context: (i) *context-free*: the code is inserted at the source point of the generation

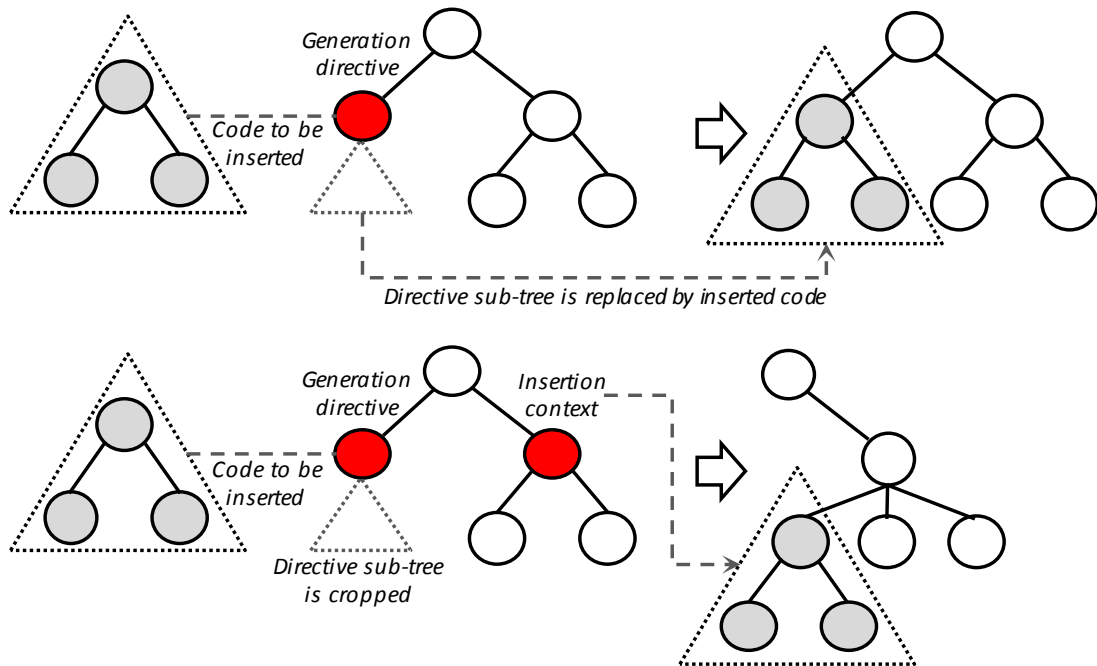


Figure 3.1 – Context-free (top) versus context-sensitive (bottom) code generation

directive; and (ii) *context-sensitive*: information on the current AST context is provided to the generation directive enabling code insertion at any AST locations reachable by the supplied context.

The context-free case is commonly called splicing or inlining and covers all cases where the generator logic does not require awareness of the code insertion context. In this case, the generator operates only on its arguments, if any, and produces a source fragment that is inserted by replacing in the AST the original generator directive. Context-free generation is very common in metalanguages, while frequently the only available option as in *Lisp*, *Scheme*, *MetaML*, *Metalua* and *Converge*.

The context-sensitive case is more general, accounting to all scenarios where the generator logic needs to decide the actual code insertion context. Additionally, the generator may insert code fragments at multiple different locations, not merely in a single context. Typically, the latter involves an AST search by the generator logic in order to locate the appropriate target contexts. Examples of context-sensitive transformations relate to meta-attribute definitions of *Nemerle* and the built-in *context* variable in *Backstage Java* [Palmer] providing access to the surrounding AST.

3.1.3 Composing and Generating All Language Constructs

While metaprograms eventually generate code, they always reflect some kind of source fragment composition logic according to particular design demands. Usually reuse is the primary motivation leading to composition and is frequently practiced by designing code skeletons or templates. In such a typical scenario, reused code clearly concerns the entire range of language constructs, while recurring code patterns become code skeletons with composition and insertion applied by metaprogramming. Now, once this type of reuse is anticipated for normal programs, there is no particular reason to be excluded for metaprograms.

In other words, metaprograms are programs too, thus deserving all features available to normal programs, including the ability to reuse any repeating metacode. For the latter it is essential that the metalanguage enables expressing and composing metacode as with normal code. In conclusion, we need to ***enable all kinds of metatags, including generator directives, to be freely quasi-quoted, manipulated and composed in the form of ASTs***. To illustrate this requirement we provide a simple

example from a text-based macro system, in particular the C preprocessor [Kernighan] where the feature is missing. In particular, macros generating further macro definitions are disabled. For instance, consider the following C macro and its use below:

```
#define SINGLE_ARG_MACRO_GENERATOR(name, arg, replacement) \
    #define name(arg) replacement
SINGLE_ARG_MACRO_GENERATOR(SQR, x, (x) * (x))
```

After the preprocessing stage, the resulting source text is as follows:

```
#define SQR(x) (x) * (x)
```

The latter is invalid as pure C code and a compile error is caused. Apparently, this problem is resolved with multiple preprocessing stages instead of a single one.

Another requirement relates to the practical limitations of quasi-quoted code to handle more comprehensive scenarios of code composition. In particular, quasi-quotes express code fragments with a constant structure, known at compile-time. They cannot express structures being the outcome of computation, such as *if* statements with a variable number of *else if* clauses. To allow generating such dynamic patterns the metalanguage should ***provide extra facilities for manipulating AST values including methods to traverse ASTs***. This may be achieved either through extra custom constructs such as algebraic data types for trees in Metalua, or via special library functions like the *ITree* functions of the *Compiler.CEI* interface in Converge.

Finally, a known issue related to generating code that introduces names is *variable capture* [Kohlbecker]. It concerns the potential of name conflicts between the inserted code and the code already available at the insertion site. The earliest approach to eliminate such conflicts was introduced in Lisp with the mandatory use of *gensym* in all macros involving temporary variables. Later, the problem was better addressed in Scheme through hygienic macros showing the importance of code generation without having to explicitly address potential name collisions.

However, there are still cases where variable capture without automatic renaming is indeed the desired effect, for instance when separately generating code for definitions and code for their actual use. In such scenarios we should enable programmers conditionally disable the hygienic behavior and preserve the original names in the

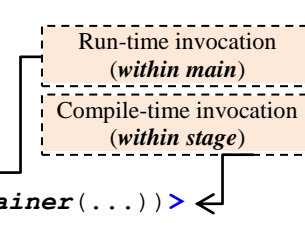
generated code. In conclusion, it is important to *support both cases by enabling the selection of either hygienic generation or name capture*.

3.1.4 Sharing and Separation of Concerns among Stages and Main

In general, all stages share the common role of transforming the source code of their embedding program. In particular, outermost stages transform the source code of the main program, while inner stages transform the source code of their enclosing stage. To realize the required transformations, stages will typically implement functionality for creating, editing and inspecting the necessary source fragments. Since similar types of generated source patterns may well reappear, *reuse and sharing of functionality across stages is prominent*.

Besides source code manipulation, stages as programs will require all sorts of utility functions commonly needed in normal programming. When such functionality is also required in the main program, *sharing between stages and main is inevitable*. In compile-time metaprogramming the latter means such functionality is available both at runtime, by main, as well as during compile-time evaluation, by stages. For example, consider the following Converge example, where functionality for some custom data containers is shared between runtime and compile-time evaluation.

```
func CreateAndPopulateContainer(...):
...
func CompileTimeCalculation(container):
...
func main():
  container := CreateAndPopulateContainer(...)
  $<CompileTimeCalculation(CreateAndPopulateContainer(...))>
...
```



Alternatively, the shared container functionality can be better organized as a library deployed both by main and stages. But again, it is not always a best practice to produce a library just to reuse some common code between stages and main. Thus, the language *should offer both options, by enabling code sharing and library deployment*, while letting programmers choose the one better fitting a situation.

Besides any possibly shared functionality between stages and main, each should remain a distinct program with its separate hidden definitions and execution state. In this sense, *encapsulation should also be supported to enable separation of concerns and thus facilitate modular staging*. An example of encapsulation is shown in the

following Metalua code, with function *CreateAST* being available only during compilation (stage) and function *Print* being available only during runtime (main).

```
-{ block: function CreateAST() return +{1} end }
function Print(x) print(x) end
Print( -{CreateAST()} )
```

3.1.5 Programming Model for Stages Equal to Normal Programs

Normal programs can be decomposed into separate modules, enabling sharing of functionality and state, while realizing a common global control flow. The present situation with stages is very different from this notion due to a custom and arguably impractical programming model commonly offered. While theoretically the existing model renders stages as expressive as normal programs, this remark refers to computability only and has little value in the software engineering quality of the model itself. More specifically, as depicted under Figure 3.2, stages are evaluated as independent transformations which operate on their input source fragments and eventually affect their enclosing program. In this sense, they resemble atomic macro

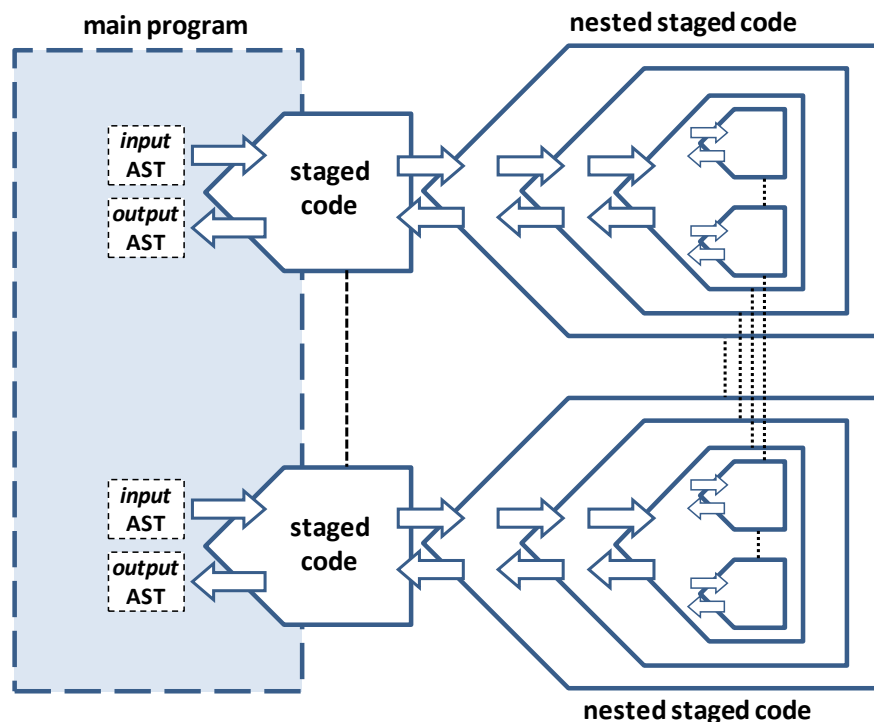


Figure 3.2 – Common evaluation of stages in popular multi-stage languages (e.g. MetaML, MetaOCaml, Converge, Metalua, etc.) and macro systems (e.g. Lisp): inside-out for nested stages, and top-down for top level stages, all as independent execution sessions. Dotted lines connect stage fragments of the same nesting level whose concatenation could comprise a single stage program.

invocations of traditional macro systems, running sequentially and within independent execution sessions, effectively operating as batches.

As shown, stages of the same nesting level always input from and output to their enclosing source text, *meaning they practically operate on the same data*. Thus, conceptually, *their concatenation may comprise a single larger stage program affecting the enclosing program*. Now, following the current practice, the evaluation of stages at the same nesting level can be actually interleaved with the evaluation of inner stages. Thus, the conceptual model of joining stages into a single program is not actually *mapped into a corresponding sequential, lexically-scoped, control flow*.

Additionally, stages are evaluated as independent programs. Then, to have some kind of state sharing across stages one should rely on custom implementation features. In particular, under interpreted language implementations, one may deploy shared global environments or dynamic scoping to feature persistent variables across the multiple interpreter invocations for stages. Not only are the latter merely implementation workarounds and not a standard property of the stage programming model, but it turns all stages, inner or outer, to a single program with a common shared state.

In general, the notion of a single program comprising *only stages of the same nesting* enabling state sharing and common control flow is not supported. In fact, most languages with compiled stages have no state-sharing workaround similar to interpreted languages. The latter disables even very simple tasks, such as implementing conditional source code insertions relying on information produced by the evaluation of preceding stages in the source text. To our knowledge, the only language that partly supports the above notion is Metalua. In fact, Metalua allows sharing state among stage code at the same nesting, as shown by the example below.

```
function Car() return {...} end

--No extension so BasicCar is same as Car
-{block: extra = +{block:}}
function BasicCar() local car = Car() -{extra} return car end

--Add ABS, so ABSCar is Car + ABS
-{block: extra = +{block: -{extra} car.abs = function () end}}
function ABSCar() local car = Car() -{extra} return car end

--Incrementally add Turbo, so ABSTurboCar is Car + ABS + Turbo
-{block: extra = +{block: -{extra} car.turbo = function () end}}
function ABSTurboCar() local car = Car() -{extra} return car end
```

```
--Incrementally add 4WD, so ABSTurboWD4 is Car + ABS + Turbo + WD4
-{block: extra = +{block: -{extra} car.WD4 = function () end}}
function ABSTurboWD4() local car = Car() -{extra} return car end
```

In this example, there is a stage variable named *extra* carrying a code block as an AST that is added to implementations of *Car* constructor functions. The code block is initially empty, i.e. *{block:}* in Metalua, and is then incrementally extended with additional statements by the successive staged code. For this example to work, the *extra* variable should be shared across the various staged code blocks.

The latter is true in Metalua because stages are not evaluated by the original language virtual machine, but by a custom interpreter supporting dynamic scoping and a common shared state across all stage executions, including nested ones. In other words, although Lua is compiled, in Metalua stages are actually interpreted. As a result, any inner stage can access and overwrite *extra* thus breaking state encapsulation on individual stages. Moreover, Metalua adopts the common multi-stage language evaluation order where stage execution is interleaved. Thus, there is no notion of sequential control flow for stage code at the same nesting.

Stages in existing languages are commonly evaluated in a depth-first fashion with either recursive interpreter invocations or successive compilation and execution rounds. Effectively, the current prevalent models for stages are two: (i) if interpreted while offering state sharing among evaluations then the stage code collectively behaves as one big program; or (ii) if compiled or interpreted without state sharing, then staged code is totally fragmented and disjoint, executed as independent sessions. We consider these two options to be special and limit cases, severely restricting the chances for deploying common software engineering practices on stages. In this context, we argue that a programming model for stages is needed *joining staged code of the same nesting into a separate coherent program, with lexically-scoped control flow, enabling software engineering practices on stages as with normal programs*. We further emphasize the equality between programs and metaprograms by denoting *Programs = Metaprograms*, or $P = MP^1$.

¹Just provoking its importance by making it sound like the $P = ?$ NP problem

We should note that, even for the other languages discussed, an advanced user or the language developer may find a way to emulate the semantics of a lexically-scoped control flow and state sharing for staged code. In this context, *our emphasis is not put on expressiveness*, meaning we do not argue that the model cannot be implemented in any of these languages. Instead, *our focus is on the optimal delivery of the model to programmers* in the most straightforward manner, as easy as with normal programs.

3.1.6 Treating Stages as First-Class Citizens of the IDE

Currently, compiled stages are evaluated as part of the build process in a way that is transparent to programming environments. For example, there is no support for build dependencies and flags on stages as with all other programs. In this context, stages should become first-class citizens of the programming environment facilitating: (i) reviewing and browsing the code of stages and the actual program transformations they introduce; (ii) improved source editing of stages through staging-aware editing automations; and (iii) a stage-aware build process. We continue by detailing the necessity for including such features in metaprogramming environments.

3.1.6.1 Source Browsing and Editing Automations

When a program that involves metaprogramming does not behave as expected it is usually difficult to directly determine the cause. The reason could be a faulty implementation of the metaprogram, wrong deployment or even some error in the logic of the final program. Since programmers only view the original source code they cannot observe the transformations performed by the metaprogram. Moreover, the metaprogram implementation may itself be generated by another metaprogram which is never part of the main source. Converge offers a solution to this problem by relating errors to all parts of the transformation path [Tratt08], thus allowing users to debug relatively easy. We consider an alternative solution, targeting to provide programmers with a view of the source code of their metaprograms as well as the transformations they perform on the main program. Apart from debugging, such a view also allows browsing through the various source code structures, providing easy access and navigation across modules, classes, functions, variables, etc. This is especially important in cases where such code is not available in the original source but generated via metaprogramming and allows programmers better understand the structures and functionality available in the generated code.

Apart from source browsing, another IDE facility that greatly improves the software development process relates to source editing automations. Relevant editing features are referred to as *IntelliSense* are generally supported in the form of tooltips that display information regarding various language expressions and symbols (e.g. argument names, symbol types, external documentation, etc.) or in the form of auto-completion. When it comes to metaprogramming, such features are invaluable as they can provide information that is not directly available from the editing context. A metafunction invocation may introduce multiple declarations to be used in the generated code which never appear in the original source text. Nevertheless, editing in a subsequent context should provide auto-completion support for them as if they were part of the original source. An example of this functionality is depicted under Figure 3.3, where both the metaprogram (i.e. the *GENERATE_CLASS* macro) and the result of its evaluation (i.e. the generated class *X*) support IntelliSense information.

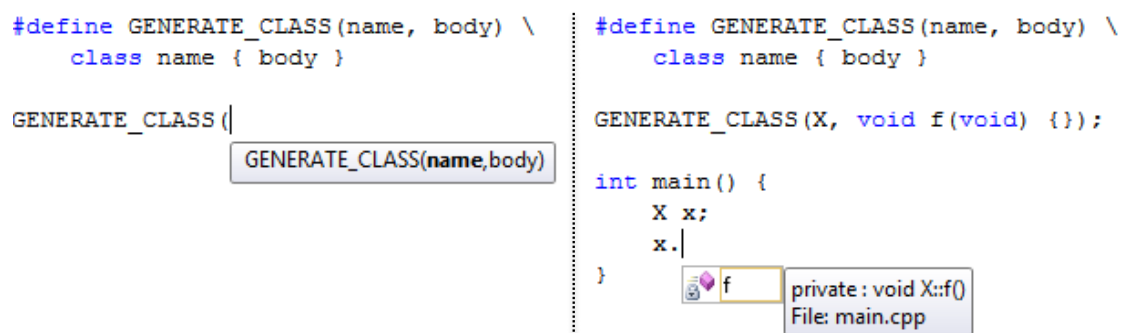


Figure 3.3 – Illustrating the support of IntelliSense information for both metaprograms (left) and their outcomes (right) when deploying CPP for code composition.

In the context of multi-staging, such functionality is even more important as small changes in the editing context (e.g. inside or outside a staging annotation) may result in a different stage and thus different set of visible symbols. In this direction, a programming environment should *produce symbolic information for staged definitions* being utilized to offer IntelliSense features as for non-staged definitions.

3.1.6.2 Build Tools

While metaprograms are encapsulated in a main program, they may require external libraries or compile flags that vary from those required by other metaprograms or the main program. For example, consider a metaprogram generating code for a GUI application. The main program will typically require a graphical library. However,

such a library is likely not needed in the generator metaprogram. Similarly, while most metaprograms will need to utilize an AST library, the main program will usually not. Consequently, programmers should be allowed to specify custom build options on metaprograms as they can on normal programs.

Besides build flags, typical build dependencies may emerge on stages too, that should be handled similarly to normal program, i.e. building any dependencies prior to stage compilation. For this to work on stages, we need to build the deployed modules prior to stage compilation, all during the compilation of the main program. But the actual build process is not handled solely by the compiler since it requires information present in the build system of the programming environment. Practically, this implies *interplay between the compiler and the build system to build stage dependencies prior to actual stage compilation*. To avoid rebuilding if stages are up-to-date, checking of respective stage binaries and their dependencies is also required.

3.2 Review

We study representative and popular multi-stage languages with respect to the identified integrated metaprogramming requirements and compare them to our approach. While we primarily focus on compile-time staging, we also discuss runtime staging within either compiled or interpreted implementations. The comparative summary is provided under Table 3.1 and shows that the requirements are only partially met by existing multi-stage languages.

Table 3.1 – Comparison of languages regarding the requirements for integrated metaprogramming. The symbol ↓ means that the feature is offered by the language with certain limitations (more details in the language-specific discussion sections).

		Compiled							Interpreted		
		C++ Templates	Template Haskell	Nemerle	Converge	Metalua	Groovy	Delta	Common Lisp	Scheme	MetaML
Exploiting Normal Language Features & Tools	Language	×	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Compiler	×	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Execution System	×	✓	✓	✓	×	✓	✓	✓	✓	✓
	Debugger	×	×	↓	×	×	✓	✓	↓	↓	×
Context-Free & Context-Sensitive Generation	Context extraction and editing	×	×	✓	×	×	✓	✓	×	×	×
	Inlining	✓	✓	✓	✓	✓	×	✓	✓	✓	✓
Composing & Generating All Language Constructs	Generation of staged code	×	×	×	×	×	×	✓	✓	✓	✓
	Quasi quotes	×	✓	✓	✓	✓	✓	✓	✓	✓	✓
	AST manipulation	×	✓	✓	✓	✓	✓	✓	✓	✓	×
	Hygienic names	×	✓	✓	✓	↓	×	✓	↓	✓	✓
	Capturing names	✓	✓	✓	✓	✓	✓	✓	✓	✓	×
Sharing & Separation of Concerns among Stages & Main	Functionality and state for stages only	✓	×	✓	×	✓	✓	✓	✓	✓	×
	Functionality shared across stages and main program	×	✓	↓	✓	↓	↓	✓	✓	✓	✓
Programming Model for Stages Equal to Normal Programs	Common state across distinct meta-code fragments	×	×	×	×	↓	×	✓	✓	✓	×
	Typical control flow across distinct fragments	×	×	×	×	×	×	✓	✓	✓	×
Stages as First-Class Citizens of the Programming Environment	Source browsing for stages	✓	↓	✓	×	×	↓	✓	×	×	×
	Source editing for stages	✓	↓	×	×	×	↓	×	×	×	×
	Build system for stages	×	×	✓	×	×	✓	✓	×	×	×

3.2.1 Compile-Time Metaprogramming

3.2.1.1 C++

C++ support for metaprogramming is based on its template system that is essentially a functional language interpreted at compile time [Abrahams][Veldhuizen96]. C++ templates do not offer the main C++ language features nor share its runtime libraries and debugging facilities. Metaprogramming is achieved by instantiating the template code with concrete types, so there is no notion of code expressed in AST form, thus disallowing any possibilities for code traversal or manipulation and there is no way for a template to compose another template. Additionally, templates cannot store or share any state and the programmer has no control on the flow of their evaluation.

Finally, regarding IDE support, programming environments like Microsoft Visual Studio provide browsing and editing features for code resulting from template instantiations. However, as previously discussed, C++ templates do not allow freely generating source code, but allow only instantiating skeleton classes and functions by filling the gaps using the supplied type arguments. Thus, providing source browsing and intelligent editing features is easier compared to general stage evaluation.

3.2.1.2 Template Haskell

Template Haskell [Sheard02] supports compile-time metaprogramming facilities through quasi-quotes and splicing. It reuses most aspects of the normal language without however providing debugging support for stages. It supports custom AST manipulation and allows generating names with either hygienic or capturing style. It also reifies the compiler's symbol table, thus enabling programmers querying the current state of compilation. Additionally, it allows querying the context through *reifyLocn* thus enabling context-dependent generation. However, it does not support splices that generate additional splices, since the splice itself is not an *Expr*.

All declared functions are available for both runtime and compile-time computations, but it is not possible to define functions used only during compile-time stage evaluation. Additionally, no function defined in a module can be used by splices in the same module; for functions to be used in a splice they must be placed within an imported module. Moreover, splices are evaluated separately, meaning there is no notion of state sharing or lexical control-flow sequence linking them. Finally, *Leksah*

[Nicklisch], a Haskell IDE offering some support for Template Haskell, does provide some browsing and editing support for stages. In particular, any declarations introduced by top level splices are visible in the source browser and are utilized by the auto-completion system during editing. However, this information is not updated during editing but requires the source file to be compiled first, meaning the symbolic information is supplied to the editor as a product of the compilation process.

3.2.1.3 Nemerle

Nemerle [Skalski04] supports metaprogramming through its macro system. As previously discussed, Nemerle macros are implemented as compiler plugins that have to be implemented separately, built as typical dynamically imported libraries (dlls). They are loaded on-demand during compilation of any source file that invokes them. Macros can be either invoked like functions to generate code during compile-time, or they can be linked (called meta-attribute definitions) to a variety of constructs like classes and methods to enable context-sensitive source code generation.

Since Nemerle macros are dynamically linked libraries it is possible to debug them using the original .NET CLR debugger. However, the debugging process is rather awkward since it requires setting the programming environment to run the Nemerle compiler in debug mode, as opposed to the user program. Then, tracing macro code is possible during the compile session. Additionally, Nemerle macros as any other dynamic libraries can share functionality and exchange state. However, such state sharing requires macro library dependencies, thus restricting modular scenarios where the shared state needs to be propagated across independently defined macros. Also, there is no explicit notion of a lexical control flow among macro invocations of the same source file, since no other stage definitions besides direct macro invocations are supported. Finally, macros cannot generate macros, thus attempting to declare the following simple macro yields a compile error.

```
macro Generator() { <[decl: macro Identity(x){x}]> } // Compile error
```

Since in Nemerle macros are not staged, but are separately edited and built, the normal non-staged source browsing facilities and build system apply (NemerleStudio or Microsoft Visual Studio with Nemerle plugin). Regarding staging, editing tooltips are supported for macros with point of definition, possible associated keywords, and

its actual evaluation result. However, there is no parameter help for macro invocations or syntax hints in case of a syntactic extension macros.

3.2.1.4 Converge

Converge [Tratt05][Tratt08] is a dynamic class-based language that allows CTMP in the spirit of Template Haskell. It reuses the original language features, compiler and execution system for metaprograms, while it currently offers no source-level debugger to judge if it is reusable on stages as well. However, since the language offers an underlying infrastructure to introduce debugging frontends it could enable the implementation of a reusable debugger frontend. ASTs can be created and manipulated either via library functions or through quasi-quotes. Additionally, generated ASTs may encompass either *alpha-renamed* (i.e. hygienic) variables or dynamically-scoped names to support variable capture. Code generation is allowed through splicing at various program locations. However, there is no support for context-sensitive insertions, neither for generation of splices thus disabling metagenerators. By design, Converge makes no scope-related distinction between normal functions and metafunctions, so all user-defined functions are eligible to be included within both stages and the final program (in fact only the minimal subset of functions are included in each stage). However, it is not possible to explicitly state that a function is only visible for compile-time computations. Finally, concerning stage evaluation, splices are treated as separate temporary modules, being independent of other splices, thus sharing no state or common control flow. For instance, assume the following hypothetical example which is not possible in Converge. The first splice declares a stage variable x and the second tries to access it. The special *pragma* splice $\$p<...>$ of Converge is here used to ignore the result of the splice expression.

```
import Sys
$p<x := 1>
$p<Sys::println(x)> // Compile error: Unknown variable 'x'.
```

The first splice alone would compile normally, with its evaluation declaring and initializing the stage variable x . Actually, if we printed its value within the first splice the result would be 1. However, once the second splice is put, a compilation error is caused, indicating that no variable x exists. As mentioned, this is due to the fact that splices are separate modules, meaning that the second splice will not find the declared variable x introduced by the first one.

3.2.1.5 MetaLua

Metalua [Fleutot07a] supports stages with the concept of separated meta-levels, allowing shifting between them using special syntax. Metacode directly embedded in the main program is referred to as level *zero*, while nested metacode takes the level of its enclosing metacode minus one. Thus, levels are numbered as 0, -1, -2, etc., with innermost levels attributing to the smallest level number. Evaluation of nested metacode is performed inside out, with inner levels always preceding the outer ones. Since metacode is evaluated top-down for level zero, and inside-out for negative levels (nested), the execution of stages is interleaved, thus supporting no notion of a lexically-scoped sequential control flow.

Sharing of functionality and state across stages is supported. This is due to a custom metacode interpreter that performs the evaluation of all stages in Metalua supporting dynamic scoping. However, such state sharing concerns all meta-levels. As a result, different meta-levels, although strictly separated and encapsulated, may access and affect the state of other meta-levels. The latter breaks encapsulation and seems more of an implementation artifact, inherent in the stage interpreter, rather than design intent. For instance, in the following example, all references to *x*, whether of stage 1 or stage 2, bind to a single stage variable, resulting in the following output: *nil, 2, 3, 1*.

```
x = 1
-{ block:
  print(x) -- uninitialized stage variable x so prints nil
  x = 2 }
-{ block:
  -{ block:
    print(x) -- binds to the earlier stage x so prints 2
    x = 3 }
  print(x) -- x retains the value of 3 above thus prints 3
}
print(x) -- binds to main program x so prints 1 (at runtime)
```

Metalua exploits all main language features, making them available in staged code, but it uses a custom interpreter implementation for stages and it does not support metacode debugging. Also, while it offers visitors and manipulators for ASTs, it forbids introduction of meta-levels programmatically, thus disabling metagenerators. Finally, generated variables are dynamically scoped, meaning they are subject to variable capture, while a hygiene library is offered utilizing *gensym* for hygienic macros.

3.2.1.6 Groovy

Groovy [Subramaniam] supports compile-time metaprogramming through AST transformations. The latter are defined as normal classes implementing the *ASTTransformation* interface and can be applied on other classes through annotations. The class annotations specify the transformations that will be invoked by the compiler while the *ASTTransformation* interface essentially provides an entry-point to obtaining and manipulating the AST of the target class. Global transformations are also supported by supplying an *ASTTransformation* subclass to the compiler that will apply it on the entire syntax tree of the code being translated, but their application is separated from the language and involves additional compilation parameters.

Groovy metaprograms reuse the language compiler and runtime system, while it is possible to debug local transformations directly from the IDE (e.g. in IDEA [JetBrains]). Additionally, since the transformations are normal Groovy code, the source browsing and editing facilities of the language can be directly deployed. However, there is no such support for the transformation outcomes: symbols resulting from transformations are invisible to the source browser and the auto-completion tool. Finally, all build flags and rules apply on transformation classes as well, thus allowing dependencies on other normal sources or even further transformations.

In Groovy, transformations are evaluated independently to each other and cannot share state or be orchestrated to a lexically-scoped control flow. They can share functionality with normal sources once organized and imported as separate modules. Finally, as with multi-stage languages studied, metagenerators are not supported.

3.2.2 Runtime Metaprogramming

3.2.2.1 Lisp and Scheme

The two major Lisp dialects, Common Lisp [Seibel] and Scheme [Dybvig09], support metaprogramming through their powerful macro systems. In Common Lisp, programs can manipulate source code as a data structure, thus macros may perform any data operation on code as well, offering the entire set of language constructs to express the transformation logic. Simple Scheme macros (created with *syntax-rules*) are essentially transformation procedures accompanied by a simple pattern matching sublanguage, while R6RS macros (created with *syntax-case* [Dybvig92]) are

procedural syntax transformers that allow destructuring input syntax objects and rebuilding syntax objects as output. At the implementation level, in both languages, normal code and macros share the same interpreter. However, while normal code is traced through the standard language debugger, macros require a dedicated macro stepper to trace code transformations resulting from macro invocations.

Lisp can create ASTs either through normal list processing functions or via the quasi-quoting mechanism, providing full support for traversal and manipulation. Code generation relies on replacing macro invocations with their output expressions, i.e. *inlining*, but without offering any context information. The output of macros can be extra macro definitions or invocations, meaning Lisp supports metagenerators. Regarding the generated names, Common Lisp offers name capture, while enabling hygienic macros using *gensym* calls. On the other hand, Scheme offers hygienic macros, while it offers the *syntax-case* clauses to selectively apply name capture. By default, Lisp and its dialects do not offer the notion of a coherent program collecting all macro definitions and invocations by stage nesting, as typical top-down and inside-out evaluation is applied. However, given its runtime system and its powerful reflectivity with self-interpretation, such functionality can be implemented as a custom library for macro management and evaluation. Apparently, the latter is feasible in Lisp, however, but it is rather complicated even for advanced users, making it more practical when offered as a built-in feature.

The reason for including Lisp dialects in the RTMP section is practical and relates to the way they are actually implemented. Most implementations are interpreted, i.e. executing instructions on an AST, meaning that macros are expanded along with the interpretation of normal code. As CTMP we consider languages directly compiled to byte code, intermediate code or machine code. In particular, for the former two cases we assume a comprehensive low-level instruction set with mostly general-purpose instructions. For instance, the *InterLisp* [Moore] virtual machine specification defines the vast majority of instructions to be Lisp dependent and to directly reflect language constructs. As a result, from an implementation perspective, a Lisp interpreter and a virtual machine are practically identical. Clearly, these are not options to judge, but we only mention to explain why we put Lisp dialects under the RTMP family.

3.2.2.2 MetaML, MetaOCaml and Mint

MetaML [Sheard98] is a multi-stage language that supports runtime metaprogramming based on staging annotations. It is based on ML, fully exploiting the original language, compiler and runtime system, however, not providing debugging support for stages. MetaML allows creating ASTs only through a quasi-quote mechanism and while it does not support explicit AST iteration it offers a pattern-matching search facility on ASTs. Also, while it offers hygienic names, it does not allow selective name capture. Code generation is achieved only through *inlining* (*Run* operator) with no support for context-sensitive generation. In MetaML, functions are shared across stages and main with no facility for hiding and encapsulation. Finally, stages do not actually share common state, other than possible cross-stage persistent [Taha97] values, while they are evaluated with the typical inside-out and top-down order, meaning staged code of different nesting levels can be also interleaved in this language.

MetaOCaml [Calcagno01][Calcagno03] is a metaprogramming extension of OCaml and is essentially a compiled dialect of MetaML. Mint [Westbrook] extends Java with the three standard multi-stage constructs, namely *brackets*, *escape* and *run*, constituting an application of these concepts in an imperative language with a compiled implementation. As such, they both share the same properties as MetaML with respect to the identified requirements.

Chapter 4

Metalanguage

"To the designer of programming languages, I say: unless you can support the paradigms I use when I program, or at least support my extending your language into one that does support my programming methods, I don't need your shiny new languages."

-Robert Floyd

We propose a new programming model for stages that we call *integrated* because it allows software engineering of metaprograms in a way similar to normal programs. Overall, the generative nature of metaprograms is treated as any other functional characteristic that programs may have, meaning no methodological separation between the two worlds is necessary. In this model, independent snippets of stage code at the same nesting are treated as a unified program, with a lexically-scoped control flow, shared program state, and the scoping rules of the main language. Additionally, all normal language features are available in implementing stages.

We continue by firstly elaborating on the programming model. Then, we brief our metalanguage constructs to support the model and discuss the semantics regarding the assembly of stage snippets in order to form integrated metaprograms. Finally, we show the expressiveness of our model in comparison to the prevalent existing model and discuss the tradeoffs involved in choosing one model over the other.

4.1 Integrated Model

As already mentioned, an integrated metaprogram is composed by the concatenation of stage code at the same stage nesting with their order of appearance in the main source. Due to this assembly, their evaluation is essentially the sequential execution of their constituent source fragments thus denoting a lexically-scoped control flow sequence within the integrated metaprogram. Since the concatenated stage fragments may encompass generative directives, an integrated metaprogram behaves as having multiple input and output locations within its *enclosing program*. We use here the

term enclosing program and not just main program because for nesting levels above one the resulting integrated metaprograms are hosted within other integrated metaprograms. In Figure 4.1 an illustration of the integrated metaprogramming model is provided, depicting source transformations, stage assembly, evaluation order and lexically-scoped (sequential) control flow.

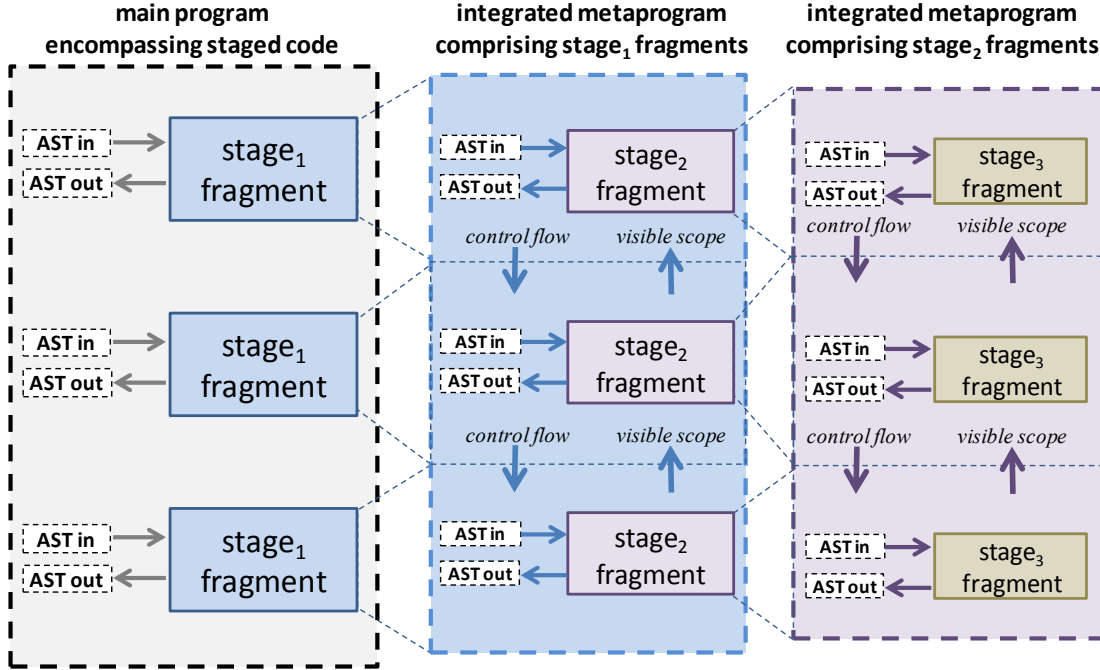


Figure 4.1 – Concept of integrated metaprograms: (i) comprising all stage fragments at the same nesting in their order of appearance; (ii) denoting a sequential control flow among stage fragments; and (iii) providing scope visibility to previous stage fragments.

In stage code any feature available in normal programs can be used, like performing typical file I/O, launching GUIs to possibly interact with programmers in tuning code generation behavior, handling network connections and communication, loading dynamically linked libraries, and so forth.

The integrated metaprogramming model compared to fragmented stage code reflects a fundamental methodological shift concerning transformations. In particular, we treat transformations as any other program function. Effectively, since stage fragments at the same nesting are related by transforming the same enclosing program, it seems an unreasonable decision to physically separate them into distinct programs or modules. Overall, segregating the stage fragments of the same enclosing program serves no

particular goal and only complicates the engineering of metaprograms. In summary, the following rounds are repeated for stage evaluation until no stages exist:

1. Determine innermost stage nesting level
2. Assemble integrated metaprogram for this nesting level
3. Build and execute

It should be noted that the result from the evaluation of each integrated metaprogram is a transformed version of main called *intermediate main*. Besides the first round using the original main, all the rest collect stage code from the current intermediate main that is then transformed to the next one. Eventually, the intermediate main from the last evaluated metaprogram is not staged and is called *final main*. It is compiled to binary constituting the output of the entire multi-stage compilation process.

From the above process for stage evaluation it is evident that our proposed model reflects a different evaluation order compared to the traditional practice of top-down and inside-out evaluation of current multi-stage languages. For example, consider the following nested staged code, where f_1 , f_2 , g_1 , g_2 are staged expressions and $!$ denotes metaprogram invocations.

```
!f1( !f2() );
!g1( !g2() );
```

The traditional evaluation order in current multi-stage languages is:

```
!f2 → !f1 → !g2 → !g1
```

The evaluation order with integrated metaprograms is:

```
{ !f2 → !g2 } → { !f1 → !g1 }
```

The brackets are used to denote that enclosed staged expressions are executed as a single coherent program and not as isolated invocations. Clearly, the two orders are different. A general form of the above example, showing the evaluation order between the two approaches is illustrated under Figure 4.2.

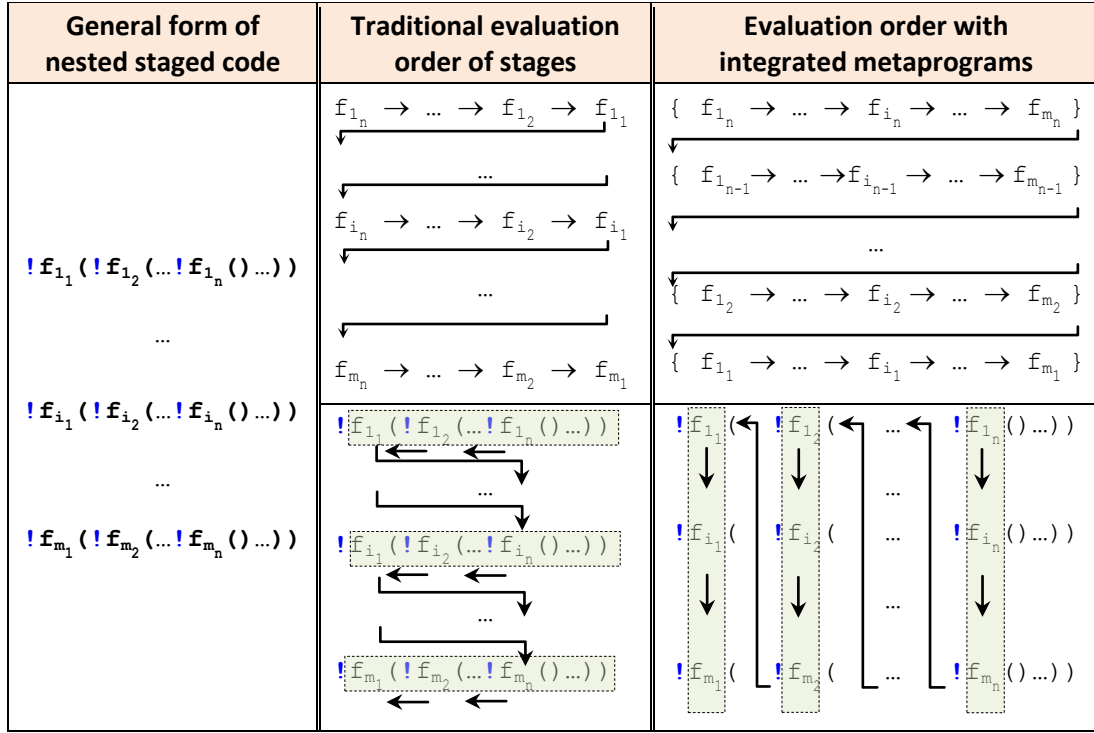


Figure 4.2 – Typical evaluation order in multi-stage languages - f_{i_j} are staged expressions with i enumerating staged code blocks and j denoting the stage nesting in the respective block.

4.2 Syntax and Semantics

As previously mentioned, our work has been implemented as the multi-stage extension of the *Delta* language [Savidis05], [Savidis10] which is untyped object-based, compiled to byte code and run by a virtual machine. In this context, the entire set of main language features are available in staged code, while only the language compiler has been extended to accommodate staging functionality. The original virtual machine has not been modified, while it is directly deployed for stage evaluation. In the following sections we briefly outline the staging syntax and semantics for our implementation of integrated metaprograms.

4.2.1 AST Tags

Such tags allow directly converting source text into ASTs, *involve no staging at all*, and are translated into calls that create ASTs by parsing source text or combining other ASTs together. This is the same approach followed by Template Haskell and Converge and could in fact be adopted to introduce similar tags in languages with no compile-time metaprogramming as extra syntactic support on a reflection library (runtime parsing, translation and execution). Besides *quasi-quotes* and *escape* that

appear in many multi-stage languages, we also introduce *delayed escape* which is essential to metagenerators.

Quasi-quotes (written `<<...>>`) may be inserted around definitions, such as expressions, statements, functions, etc., to convey their AST form and are the easiest way (not the only one) to create ASTs directly from source text. For instance, `<<1+2>>` is the AST for the source text `1+2`, while `ast_make_const(3.14)` produces the AST for the numeric constant `3.14`. Quasi-quotes are allowed to be nested arbitrarily, something useful in implementing *metagenerators*. For example, `<< <<1+2>> >>` is a nested quasi-quoted expression that represents the quasi-quoted expression `<<1+2>>`. For variables present within quasi-quotes we identify three possible binding policies: (i) *late binding*; (ii) *early binding*; and (iii) *no binding* (or alpha-renaming).

Late binding means that variables within quasi-quotes are scoped in the context where the respective AST is finally inserted. For instance, `<<x=1>>` does not bind to any `x` visible at the quasi-quote location. In the Delta language, variables are lexically-scoped and are declared-by-use the first time a name is met. Thus, if no `x` is defined at an insertion point a new `x` is introduced by the assignment.

Early binding means that the variable within the quasi-quotes refers to a symbol syntactically visible at the scope of the quasi-quote declaration. Such a policy is used for example in C++ templates, where a name present within a template body is resolved by first looking in the context of the template declaration instead of the context of the template instantiation. In general this is a useful binding policy; however in the particular implementation in Delta, it does not fit well with the language semantics. Firstly, there is the issue of *scope extrusion*, in which a symbol within quasi-quotes referring to a specific program variable may be used at a context where the variable it refers to is out of scope. Then, it is possible for the symbol within quasi-quotes to bind to some inner function allowing it to be used at a place where it should not be visible, thus breaking encapsulation. The same applies if we allow referring to non-exported functions of a module potentially allowing other modules to refer to them. Practically, the only valid usage involves binding to an exported top level (i.e. global) function so that it can be used from other modules or

from a context where some local definition would otherwise shadow the global one, preventing a default late binding approach from referring to the desired symbol. Nevertheless, the same functionality can be achieved by simply supporting fully qualified names for symbols within quasi-quotes. In this sense, an early bound reference to a top level function can be achieved by explicitly specifying the module in which the function is present, denoted as `<<module::f>>`.

The no binding policy is used to solve the issue of variable capture and supports hygienic macros by introducing alpha-renamed variables, i.e. variables that are given automatically contextually-unique identifiers. In particular, we use the notation `<<$x>>` to specify that `x` will be given a fresh unique name at the insertion context. In most metalanguages such hygienic behavior is active by default with name capture (i.e. late binding) enabled only through special syntax.

We have purposefully chosen such an inverse activation policy since we consider it to more frequently fit the actual use of metaprograms. More specifically, metaprograms may produce: (i) complete named elements such as classes, functions, methods, constants, namespaces and generics that may be directly deployed; (ii) template code fragments to be filled-in with other code fragments; (iii) other non-template code fragments that may be further combined.

In case of named elements, the supplied name will be directly used for deployment, thus name capture is the only way. When generating non-template code fragments, those may be further composed together or inserted in templates. In this case, the statements of such code fragments may erroneously capture earlier or outer variables. The latter is avoided easily in the respective generator by always declaring generated variables as local and enclosing related statements in blocks. Finally, in case of templates, it is possible that inserted code fragments may undesirably capture names in the template itself. This is the only case where the template generator should force hygiene for template variables. Overall, based on the previous remarks, we considered that for most scenarios name capture would suffice and for the template cases hygiene may be deployed where required.

Normal Escape (written $\sim(\text{expr})$) is used only within quasi-quotes to prevent converting the source text of expr into an AST form by evaluating expr normally. Practically, escape is used on expressions already carrying AST values which need to be combined into an AST constructed via quasi-quotes. For example, assuming x already carries the AST value of $\langle\langle 1 \rangle\rangle$, the expression $\langle\langle \sim x + 2 \rangle\rangle$ evaluates to $\langle\langle 1 + 2 \rangle\rangle$. The latter also applies in nested quasi-quotes, meaning the expression $\langle\langle \langle\langle \sim x + 2 \rangle\rangle \rangle\rangle$ evaluates to $\langle\langle \langle\langle 1 + 2 \rangle\rangle \rangle\rangle$. Additionally, we also support the escaped expression to carry scalar values like number, boolean or string (i.e. *ground* values). In this case, the value is automatically converted to its corresponding AST value as if it were a constant. For instance, if x is 1, then $\sim x$ within $\langle\langle \sim x + 2 \rangle\rangle$ will be converted to the AST of value 1, or $\langle\langle 1 \rangle\rangle$, thus $\langle\langle \sim x + 2 \rangle\rangle$ evaluates to $\langle\langle 1 + 2 \rangle\rangle$.

Delayed Escape (written $\sim \dots \sim(\text{expr})$) is used when escape evaluation should be deferred, something common in metagenerators. The number of tildes is the initial *nesting* which for normal escapes is one. Then, escape evaluation, being performed when quasi-quotes are evaluated, is applied as follows:

$$\text{eval}(\text{escape}(n, \text{expr})) = \begin{cases} \text{escape}(n - 1, \text{expr}), & n > 1 \\ \text{expr}, & n = 1 \end{cases}$$

Notice that the previous evaluation is not recursive – it returns either the escaped expression or a new escape with decreased nesting. Practically, delayed escapes will be at some point inlined into generated quasi-quotes. The following examples simply outline the behavior of delayed escape (*gen* denotes code generation with an AST parameter). Later, once the details of the staging tags and integrated stage assembly are explained, more elaborate examples with metagenerators are discussed showing the importance of delayed escapes.

- Writing $\langle\langle \langle\langle \sim \sim x \rangle\rangle \rangle\rangle$ represents the AST of $\langle\langle \sim x \rangle\rangle$
- With $y = \langle\langle \sim \sim x \rangle\rangle$ the expression $\text{gen} \langle\langle \langle\langle \sim y \rangle\rangle \rangle\rangle$ yields $\langle\langle \sim x \rangle\rangle$

The introduction of delayed escapes allows generating escapes and preserve syntactically their presence within quasi quotes. The latter is in contrast to the single escape preserving the value it carries. One could also think of a library function like

`esc(ast, n)` to generate a chain of a n parent escapes on the supplied syntax tree. Now, while the latter would produce tree forms identical to the delayed escape ones, without $\sim^n(\text{expr})$ in the language syntax the resulting trees would be syntactically ill-formed. In fact, in our implementation all syntax trees composed through library functions pass internal syntactic validity checks, meaning the output of such library function would be directly rejected. We consider delayed escapes to serve a purpose similar to quasi-quotes: one may live without them but using them makes life easier.

The fact that the previous tags do not introduce staging as such, but are essentially facilities for AST manipulation is depicted under Table 4.1. As shown, quasi-quotes are shortcuts for AST creation (`ast_create`), the latter in our language handled with internal parser invocations. Similarly, escapes (`ast_escape`) involve AST composition operations, again without staging, and are only invoked for normal escapes.

Table 4.1 – Code generation examples for quasi-quotes and escapes, showing that they do not involve staging.

AST tag expressions	Respective intermediate code
<code><<1 + g()>>;</code>	<code>ast_create \$0 "1 + g()"</code>
<code><<~(f(x)) + 2>>;</code>	<pre> param x call f getretval \$0 #carries f(x) ast_create \$1 "~(f(x)) + 2" ast_escape \$1 \$0 #inserts f(x) </pre>
<code><< << ~~x >> >></code>	<code>ast_create \$0 "<< ~~x >>"</code>
<code><< << ~~x + ~y >> >></code>	<pre> ast_create \$0 "<< ~~x + ~y>>" ast_escape \$0 y #inserts y </pre>
<code><< f(~a, ~b) >></code>	<pre> ast_create \$0 "f(~a, ~b)" ast_escape \$0 a #inserts a ast_escape \$0 b #inserts b </pre>

4.2.2 Staging Tags

Staging tags generally imply compile-time evaluation of associated source code, and are essential in supporting compile-time metaprogramming. Syntactically, they define the boundaries between staged code fragments and also introduce stage nesting, also known as metalevel shifting. Besides *inline* (generation) and *execute* (metalevel shifting), appearing in various metalanguages, we introduce *define*. We will show that the latter is required for languages where *execute* cannot syntactically represent both

block-scoped statements and program-scoped definitions. For instance, interface definitions and import directives in Java are only globally defined, while statements can only be locally defined in blocks. Thus, there can be no appropriate common non-terminal for both which *execute* could adopt to address both. We will further discuss this matter below, after first detailing the staging tags syntax and semantics.

Inline (written `!(expr)`) is staged code evaluating `expr` (whose value must be of an AST type) into the enclosing program by substituting itself. Inline tags within quasi-quotes are allowed, and as all other quasi-quoted expressions, are just AST values that are not directly evaluated. It is allowed for expressions carrying an AST representing an inline directive to be inlined, meaning generation directives may generate further generation directives, thus supporting metagenerators. The latter, besides nested stages statically defined via explicit staging tags allows any number of stages to be dynamically introduced by metaprograms themselves.

As an extreme example, the following inline directive (the second one) repeatedly reproduces itself (using the first one that is quasi-quoted), causing endless staging.

```
function f() { return <<!(f())>>; }
!(f());
```

With a small change, the same example works in a way that the inline directive reproduces itself a controlled number of times. The `nil` value shown in the example denotes an empty AST value that essentially replaces the generator with no code.

```
function f(n) { return n < 1 ? nil : <<!(f(~(n-1)))>>; }
!(f(10));
```

Execute (written `&stmt`) defines a staged `stmt` representing any single statement, local definition or block in the language. Any definitions introduced are visible only within staged code. Execute tags can also be nested (e.g. `&&stmt`), with their nesting depth specifying the exact compilation stage they will appear in. Essentially *execute* is similar to Metalua's metalevel shifting construct `-{...}`. For example, the Delta code `&std::print(1)` is equivalent to Metalua's `-{ print "1" }` while `&&std::print(2)` is equivalent to `-{-{ print "2" }}`. Additionally, execute tags can be quasi-quoted and be converted to AST form, meaning their inlining will introduce further staging.

The following is a simple example (adopted from [Czarnecki]) combining the use of *inline* and *execute* tags. The function `ExpandPower` creates the AST of its `x` argument being multiplied by itself `n` times, while `MakePower` creates the AST of a specialized power function. It should be noted that in Delta, function definitions are syntactically statements, thus allowed within *execute* tags. As shown, the code resulting from this program encompasses only an assignment of the generated anonymous function.

```
&function ExpandPower (n, x) {
    if (n == 0)
        return <<1>>;
    else
        return <<~x * ~(ExpandPower(n - 1, x))>>;
}
&function MakePower (n) {
    return << (
        function(x) { return ~(ExpandPower(n, <<x>>)); }
    )>>;
}
power3 = !(MakePower(3));
power3 = (function(x) { return x * x * x * 1; });
```

Define (written `@defs`) allows introducing stage `defs`, the later syntactically representing any *global* program unit in the language (e.g. global definitions). Define tags may be nested (e.g. `@@def`) with the nesting depth specifying the stage the *defs* will appear in, being analogous to nested *execute* tags.

This tag is only needed for languages where there is a syntactic distinction between global and local definitions. Thus, in languages such as Lua, JavaScript or Delta the latter is not needed since global and local elements are not syntactically separated. There, *execute* can do what *define* is supposed to offer. But *define* is required in languages like C++, Java or C# since there are differentiations as to what can be defined locally or globally.

Now, why *define* becomes necessary in such language case and what actual metaprogramming need it serves becomes clear with an example. Consider the staged code of Figure 4.3 (top left part), having stage nesting 1, in a hypothetical meta-C++ language adopting our staging tags. When no *define* is available, stage code adopts *execute* and *inline* tags, our example using just *execute*. The integrated metaprogram is assembled from all *execute* snippets into the `stage_1()` function of Figure 4.3

(right part). Notice two function definitions in *execute* tags, namely *f* and *Load_B* (shaded areas, top left part of Figure 4.3) whose functionality is required in stage code. The problem is that their concatenation into *stage_1()* constitutes illegal C++ syntax as no local function definitions are allowed (shaded areas, right part of Figure 4.3).

<i>meta-C++ staged code:</i>	<i>assembled main stage block:</i>
<code>& int f(...) {...}</code>	<code>class C {...};</code>
<code>& int x = f(...);</code>	<code>void stage_1(...) {</code>
<code>class C {...};</code>	<code>int f(...) {...}</code>
<code>& static B* Load_B(C* c) {...}</code>	<code>int x = f(...);</code>
<code>& C* c = new C(...);</code>	<code>static B* Load_B(C* c) {...}</code>
<code>& B* b = Load_B(c);</code>	<code>C* c = new C(...);</code>
<code>& class A {...};</code>	<code>B* b = Load_B(c);</code>
<code>& A* a = new A(...);</code>	<code>class A {...};</code>
<i>main() function of the integrated metaprogram:</i>	<code>A* a = new A(...);</code>
<code>int main (int argc, char** argv){</code>	<code>}</code>
<code>... stage_1(...); ...</code>	
<code>}</code>	

Figure 4.3 – Without supporting *define* all stage snippets are by default collected inside the main stage block, that could cause ill-formed C++ syntax as C++ forbids local function definitions (shaded code).

Offering *define* allows the distinction between global definitions and statements, enabling metalanguages to assemble integrated metaprograms by separating **global stage definitions** from the main execution block. Using *defines* we rework our example in Figure 4.4, turning the resulting C++ metaprogram to syntactically correct.

<i>meta-C++ staged code:</i>	<i>global code comprising non-stage definitions and stage definitions from <u>define</u> tags:</i>
<code>@ int f(...) {...}</code>	<code>int f (...) {...}</code>
<code>& int x = f(...);</code>	<code>class C {...};</code>
<code>class C {...};</code>	<code>static B* Load_B(C* c) {...}</code>
<code>@ static B* Load_B(C* c) {...}</code>	<code>class A {...};</code>
<code>& C* c = new C(...);</code>	<i>main stage block comprising stage code from <u>execute</u> tags:</i>
<code>& B* b = Load_B(c);</code>	<code>void stage_1 (...) {</code>
<code>@ class A {...};</code>	<code>int x = f(...);</code>
<code>& A* a = new A(...);</code>	<code>C* c = new C(...);</code>
<i>main() function of the integrated metaprogram:</i>	<code>B* b = Load_B(c);</code>
<code>int main (int argc, char** argv){</code>	<code>A* a = new A(...);</code>
<code>... stage_1(...); ...</code>	<code>}</code>
<code>}</code>	

Figure 4.4 – When supporting *define*, only stage code from *execute* directives is collected inside the main stage block; code from *define* is assembled with rest non-stage global definitions, following their order of appearance, resulting in syntactically correct C++ code.

As shown, *define* allows programmers to explicitly instruct the stage assembler to separate specific definitions from the main stage block and place them in the global definitions section of the integrated program.

We may try dropping the extra *define* tag by extending the semantics of *execute* to automatically treat global definitions as implicit *define* directives. Now, this can be problematic in languages such as C++ where elements allowed in a global context, like classes and type definitions, may also appear locally in a block. If the goal is to locally define a class hidden outside, such as for template instantiations (being a very common practice), the class will be placed in the global section and encapsulation is broken. Consider the example of Figure 4.5 where local type definitions are provided. This example fails as staged code when implicit *define* directives are implemented due to name conflicts when locally defined types are transferred in the global section.

<i>meta-C++ staged code:</i>	<i>global code and main stage block of the integrated metaprogram:</i>
<pre> class A {...}; & if (...) { typedef pair<int,int> A; typedef list<A> ListA; ... } & while (...) { typedef list<A*> ListA; ... } </pre>	<pre> class A {...}; typedef pair<int,int> A; ←error typedef list<A> ListA; typedef list<A*> ListA; ←error void stage_1 (...) { if (...) { ... } while (...) { ... } } </pre>

Figure 4.5 – The automatic treatment of *execute* directives involving global definitions as *define* directives disables encapsulation and information hiding for element categories allowed both at global and local scope; this may cause semantic errors, such as replicate definitions (name conflicts).

4.2.3 Stage Assembly and Evaluation

The current integrated stage program is composed by considering staged code only at the innermost level, thus consisting of non-staged code. It consists of two sections, one after the other: *global area* and *main block*. The main block collects code from *execute* and *inline* directives concatenated together in the order they appear in the source text. The global area encompasses all main program definitions used by the current staged code, and also all code from *define* directives (if applicable), while preserving the relative order of appearance of concatenated fragments in the main source text. An example is provided under Figure 4.6 illustrating this process.

Sections	Main program	Integrated metaprogram
global area	<pre> other defs main_defs_0 main_defs_1 &code_1 @defs_1 main_defs_2 !(expr_0) &code_2 &code_3 other defs </pre>	<pre> main_defs_0 main_defs_1 defs_1 main_defs_2 </pre>
main block		<pre> code_1 inline expr_0 code_2 code_3 </pre>

Figure 4.6 – Illustrating the assembly of integrated metaprograms with a general example with arrows outlining dependencies – only the required definitions from the main program are included.

The stage assembly process begins by computing the maximum stage nesting with a traversal on the entire AST. This computation should be repeated at the beginning of every stage evaluation since the maximum stage nesting may be increased if the evaluation of the last stage has generated further staged code. Then we need to perform a depth-first traversal and collect source code from all staging tags in this nesting. For *execute* and *define* tags, the associated code is actually all that is needed and it can be used as it is, while cutting respective nodes from the main program AST.

However, *inline* directives require a different treatment, since merely copying the associated *expr* in the main block will not realize its expected generative behavior. The latter, as mentioned earlier, involves substitution of the *inline* node by its *expr* value. Clearly, some special invocations need to be included which will internally handle the required AST modifications. In this context, an effective approach is to adopt a library function offered by the meta-compiler that is linked only with integrated metaprograms. This ensures integrated metaprograms are syntactically just normal programs and can be compiled using the original language compiler. When running, the meta-compiler is just part of their execution environment.

In our implementation this function is called `std::inline`, with no result and a single argument being the *expr* of AST type. Then, while assembling the stage, the *expr* node is cut, leaving a leaf *inline* alone, and an `std::inline(expr)` invocation is introduced in the main block. The role of the *inline* leaf is to be a bookmark for the insertion point of its respective `std::inline` call. For this reason, an *inline* leaf is pushed on a stack, exactly after its respective `std::inline` call is placed in the main block, thus ensuring their relative orders match. Then, the

`std::inline` function simply pops an *inline* leaf from the stack and substitutes it by its *expr* argument.

Definitions from the main program deployed across stages need not be placed in external modules but are directly included in the stage assembly (reflected in the *main_defs_i* parts of Figure 4.6). Further dependencies of copied definitions should be copied as well, the process being recursive until no required dependencies are missing. In case of deployed functions they should not depend on the state of the main program, thus they may be safely copied and become an integral part of another program. In particular, for a function to be eligible for inclusion in stages, it should not: (i) contain staging tags (i.e. the entire definition should be present at the current stage); (ii) access global variables of the main program; (iii) use any closure variables; and (iv) use other functions that do not meet the same requirements. The reason for excluding the use of closure variables is that in general they may involve runtime functionality as they depend on the execution of the outer function. To determine which functions meet the above requirements, during the AST traversal we also collect information about function dependencies and use this information to extract the eligibility information. More specifically, when traversing the body of a function, upon any function usage we apply the following rules:

1. If the target function is ineligible, the current function depends on an ineligible function and is therefore ineligible as well.
2. If the target function is eligible, we cannot yet determine about the current function but have to continue the traversal; if all function dependencies are found are eligible then the current function is eligible as well.
3. If the eligibility of the target function is not yet determined, we simply mark the dependency and continue. If any dependent function later proves to be ineligible the current function also becomes ineligible. If, on the other hand, all prove to be eligible then the current function is eligible as well. Finally, if we have a cyclic dependency we continue until all dependency information about functions involved in the circle are available. Eventually, if all functions in the circle depend only on each other and not on other ineligible function, they are all considered eligible.

In terms of performance, the integrated staging approach involves a single compilation and evaluation round per stage nesting. In comparison, existing multi-

stage languages involve one round per staged code fragment. Effectively, the evaluation of integrated metaprograms is overall more efficient.

An example is provided under Figure 4.7 with a multi-stage main program, its integrated metaprograms with resulting intermediate and final main versions.

Main and intermediate versions	Integrated metaprograms
<p><i>Original staged main</i></p> <pre> &&function create_macro(name, args, val) { return << function ~name (~args) { return <<~val>>; } >>; } &!(create_macro(<<identity>>, <<x>>, <<~x>>)); &!(create_macro(<<identity>>, <<x>>, <<~x * 2>>)); x = !(identity(<<1>>)); y = !(double(<<1>>)); </pre>	<p><i>Stage nesting 2</i></p> <pre> function create_macro(name, args, val) { return << function ~name (~args) { return <<~val>>; } >>; } std::inline(create_macro(<<identity>>, <<x>>, <<~x>>)); std::inline(create_macro(<<identity>>, <<x>>, <<~x * 2>>)); </pre>
<p><i>Intermediate staged main after last stage</i></p> <pre> &function identity(x) { return <<~x>>; } &function double(x) { return <<~x * 2>>; } x = !(identity(<<1>>)); y = !(double(<<1>>)); </pre>	<p><i>Stage nesting 1</i></p> <pre> function identity(x) { return <<~x>>; } function double(x) { return <<~x * 2>>; } std::inline(identity(<<1>>)); std::inline(double(<<1>>)); </pre>
<p><i>Final non-staged main after last stage</i></p> <pre> x = 1; y = 1 * 2; </pre>	<p><i>Stage nesting 0</i></p> <p>No more staged code</p>

Figure 4.7 – Left: main with its intermediate and final versions; Right: Integrated metaprograms from the original and intermediate main versions.

Initially, the maximum stage nesting is 2, tied between the declaration and usages of function `create_macro` (prefixed by the tags `&&` and `&!` respectively). The first stage thus contains the code present within these tags (Figure 4.7, top left, highlighted with a dotted rectangle), along with the appropriate invocations to `std::inline` (generated by the inline directives) for performing the code generation. Since `create_macro` is defined at stage nesting 2 it is removed from the intermediate main, while all `inline` tags at nesting 2 are substituted by the generated code of their `std::inline` calls.

Thus, the resulting intermediate main contains the stage definitions of functions `identity` and `double` along with remaining code (i.e. the two assignments) that was part of the original program (Figure 4.7, middle left, highlighted with a dotted shape). The same process continues for the next stage. Now the stage nesting is 1, tied between the definitions and usages of functions `identity` and `double`, so the

extracted metaprogram contains their code along with the extra `std::inline` invocations. After stage execution, the function definitions of the stage itself are removed, and the *inline* directives are again replaced by the generated code of their `std::inline` calls. This results into the final main having no further stages (Figure 4.7, bottom left).

4.2.4 Enabling Metagenerators

As previously mentioned, to support metagenerators the language should enable creating ASTs with nodes representing staging tags. Simply inlining such ASTs will introduce staging. The latter can be done by supporting staging tags directly in quasi-quotes or through AST composition using a library. In general, stages may even generate code with more deep staging than themselves. For instance, consider the example shown in Figure 4.8, where `meta_gen` is a metafunction that is capable of generating code with arbitrary stage nesting, even though it is defined in the first stage with nesting 1. In our example, it is invoked twice to generate two `print` calls, at stage nesting 1 and 2 respectively. Notice that the loop assignment `code = <<&~code;>>`; essentially prepends the syntax tree carried by `code` with an extra *execute* parent tag, thus increases its stage nesting in every iteration. As a result, the two invocations return the trees `<<&std::print(1);>>` and `<<&&std::print(2);>>` assigned to stage variables `x` and `y` respectively. These trees are combined with quasi-quotes and are *inlined* to introduce additional staging.

Main program transformations	Stage metaprograms
Original staged main <pre>function meta_gen(code, n) { for (local i = 0; i < n; ++i) code = <<&~code;>>; return code; } x = meta_gen(<<std::print(1)>>, 1); y = meta_gen(<<std::print(2)>>, 2); !(<<&~x;~y;>>);</pre>	Stage nesting 1 <pre>function meta_gen(code, n) { for (local i = 0; i < n; ++i) code = <<&~code;>>; return code; } x = meta_gen(<<std::print(1)>>, 1); y = meta_gen(<<std::print(2)>>, 2); std::inline(<<&~x;~y;>>);</pre>
Intermediate staged main after last stage <pre>&std::print(1); &&std::print(2);</pre>	Stage nesting 2 <pre>std::print(2); ***This print is performed by the stage</pre>
Intermediate staged main after last stage <pre>&std::print(1);</pre>	Stage nesting 1 <pre>std::print(1); ***This print is performed by the stage</pre>
Final non-staged main after last stage No more source in main	Stage nesting 0 No more staged code

Figure 4.8 – An example where the first evaluated stage is a metagenerator. Left: main with intermediate and final versions; Right: Integrated metaprograms from original and intermediate main versions.

4.2.5 Context Sensitivity

As discussed earlier, context sensitivity is supported when the actual source code insertion point may become the outcome of a computation. Currently, the *inline* directives of multi-stage languages denote a statically defined context which is the particular location of the directive itself. To support code generation at an arbitrary context, i.e. location within the source code, a possible solution is to offer an entry point for obtaining and directly manipulating the source program AST. In this direction, and following the approach of offering the functionality as a special library function so as to allow stages to be syntactically normal programs, we propose the provision of another compile-time library function which simply returns the AST node of itself, representing its actual location in the source program. In our implementation the latter is named `std::context`. For convenience, we also offer an overloaded version that receives an AST tag and instead returns the closest matching ancestor node. This way an invocation `std::context(tag)` operates almost as it reads; it returns the AST node that matches the given tag within the invocation context.

We provide an example of context-sensitive generation for our implementation in the Delta language, where objects are created ex-nihilo via respective constructor expressions, also called object expressions. Consider an object expression in which we wish to insert *set* / *get* methods for its members. Instead of repeated *inline* directives per data member we use `std::context` to get the object expression AST, traverse it to find the data members and then: (i) directly attach to the AST the required method definitions; or (ii) produce the AST for the method definitions, and *inline* its returned value where desired. Both options for context-aware code generation are illustrated in the following example.

```
&function InsertAccessors(obj) {                                     ← obj input is an object AST node
  foreach (local attr, obj.getAttributes()) { ← iterate over object attributes
    local name = attr.getName();
    local set  = <<method ~("set_" + name) (val) { self.~name=val; }>>;
    local get  = <<method ~("get_" + name) () { return self.~name; }>>;
    obj.addMethods(set, get); ← insert set / get methods directly in object expression
  }
  return nil;                                     ← no code explicitly returned
}

&function GenerateAccessors(obj) {                                   ← obj input is an object AST node
  local result = nil; ← will hold the generated method code to be inlined in the object
  foreach (local attr, obj.getAttributes()) { ← iterate over class attributes
```



```

    local name = attr.getName();
    local setter = <<method ~("set_" + name) (val) {self.~name=val;}>>;
    local getter = <<method ~("get_" + name) () {return self.~name;}>>;
    result = <<~result, ~setter, ~getter>>;  ← combine methods in a new AST
  }
  return result;  ← all generated accessor methods are returned to be inlined by client code
}

const OBJ_TAG = "ObjectConstructor";  ← tag matching any object expression
function Point2D(x, y) {
  return [
    @x : x, @y : y,
    !(InsertAccessors(
      std::context(OBJ_TAG)
    ));
  ];
}
function Point3D(x, y, z) {
  return [
    @x : x, @y : y, @z : z,
    !(GenerateAccessors(
      std::context(OBJ_TAG)
    ));
  ];
}

```

→

```

function Point2D(x, y) {
  return [
    @x : x, @y : y,
    method set_x(val) { self.x = val; },
    method get_x() { return self.x; },
    method set_y(val) { self.y = val; },
    method get_y() { return self.y; }
  ];
}
function Point3D(x, y, z) {
  return [
    @x : x, @y : y, @z : z,
    method set_x(val) { self.x = val; },
    method get_x() { return self.x; },
    method set_y(val) { self.y = val; },
    method get_y() { return self.y; },
    method set_z(val) { self.z = val; },
    method get_z() { return self.z; }
  ];
}

```

4.2.6 Metacode Libraries

As previously discussed, a metaprogram may use any normal program feature including the deployment of other modules. In the Delta language the latter is handled through the *using* directive that specifies the module to use. In the same sense, the deployment of a module within a stage can be achieved by adding staged *using* directives with the appropriate stage nesting. For example, let's revisit the example of the previous section that automatically introduces accessor methods for its members. The function *InsertAccessors* can be placed as non-staged code within a separate module called *MetaUtils* and deployed to perform the same transformation.

```

&using #MetaUtils;
const OBJ_TAG = "ObjectConstructor";  ← tag matching any object expression
function Point2D(x, y) {
  return [
    @x : x, @y : y,
    !(MetaUtils::InsertAccessors(std::context(OBJ_TAG)));
  ];
}

```

We should note that the *using* directive actually refers to the binary file of the specified module. In this sense, even if the source file of that module originally

contained metacode, it has been already evaluated during its compilation and is no longer available in the binary form. This essentially means that metaprogram fragments of a source file cannot interact with or manipulate metaprogram fragments of another source file. The latter is only possible through a higher order metaprogram present in the same source or through external source transformation approaches.

The use of the binary file of the target module introduces an additional issue in case it is not available prior to the staged program evaluation. To allow the dependency to be resolved, we should naturally compile the target module as well before continuing with the evaluation. This issue is addressed later in section 5.2 where we discuss the interaction between the compiler and the build system.

4.3 Expressiveness

We prove that the integrated metaprogramming model is at least as expressive as the current multi-stage programming model. Effectively, using the staging tags, evaluation order and stage assembly semantics of our language we emulate the top-down inside-out evaluation order of existing multi-stage languages.

In the introduction of our model we explained the different evaluation order between current multi-stage languages and integrated metaprograms. We recall the example we used and its traditional evaluation order below:

```
!f1(!f2() );
!g1(!g2() );

!f2 → !f1 → !g2 → !g1
```

We can emulate the traditional stage evaluation sequence under the integrated metaprogramming model by modifying the example as follows.

```
!f1(!f2() );
!<<!g1(!g2() )>>;
```

The second staged expression `!<<!g1(!g2())>>` denotes a metaprogram inlining the quoted AST `<<!g1(!g2())>>` and now has stage nesting one. The key point here is that once we quasi-quote a definition we turn it to a constant non-staged AST expression. We recall our earlier discussion on tags where we mentioned that none of the AST tags are staging tags. Essentially, we *hide* any staging embedded in the quasi-quoted definitions, until *revealed* latter at some point through inlining. The maximum

stage nesting in the refined program is two and concerns f_2 in $!f_1(!f_2())$ thus the first stage consists only of the execution of f_2 . Then, the resulting program consists of $!f_1$ and $!<<!g_1(!g_2())>>$, both with stage nesting one, thus composing the next stage and evaluated together. Up to this point, the evaluation sequence is the following (blocks indicate distinct stage programs, arrows indicate order).

$$\{ !f_2 \} \rightarrow \{ !f_1 \rightarrow !<<!g_1(!g_2())>> \}$$

The expression $!<<!g_1(!g_2())>>$ is only used to generate the code whose original staging was hidden with quasi-quotes, thus revealing its staging: $!g_1(!g_2())$;

In the same way as before, this code will further evaluate as follows, with two different stage programs: $\{ g_2 \} \rightarrow \{ g_1 \}$. Overall, the resulting evaluations follow a sequence that is identical to the traditional order. This method can be generalized for any multi-stage program and may be automated in the compiler, if traditional evaluation is needed, through AST manipulation. More specifically, we can locate all top-level staged fragments and surround them with an increasing number of nested $!<< \dots >>$ tags, starting with zero. Thus, the first fragment remains as it is (zero tags), the second is put inside $!<< \dots >>$ (one tag), the third inside $!<< !<< \dots >> >>$ (two tags), and the n -th inside $n-1$ tags. This process is illustrated in Figure 4.9 and shows that the integrated model can emulate the traditional model.

General form of nested staged code	Emulating the traditional evaluation order in the integrated model
$!f_{1_1}(!f_{1_2}(!f_{1_3}(\dots !f_{1_{n_1}}())\dots))$	$!f_{1_1}(!f_{1_2}(!f_{1_3}(\dots !f_{1_{n_1}}())\dots))$
$!f_{2_1}(!f_{2_2}(!f_{2_3}(\dots !f_{2_{n_2}}())\dots))$	$!<< !f_{2_1}(!f_{2_2}(!f_{2_3}(\dots !f_{2_{n_2}}())\dots)) >>$
$!f_{3_1}(!f_{3_2}(!f_{3_3}(\dots !f_{3_{n_3}}())\dots))$	$!<< !<< !f_{3_1}(!f_{3_2}(!f_{3_3}(\dots !f_{3_{n_3}}())\dots)) >> >>$
\dots	$i-1 \text{ repetitions } !<< \dots !f_{i_1}(!f_{i_2}(!f_{i_2}(\dots !f_{i_{n_i}}())\dots)) \dots >> i-1 \text{ repetitions}$
$!f_{i_1}(!f_{i_2}(!f_{i_2}(\dots !f_{i_{n_i}}())\dots))$	\dots
\dots	$!<< \dots !<< !f_{m_1}(!f_{m_2}(!f_{m_3}(\dots !f_{m_{n_m}}())\dots)) >> \dots >>$
$!f_{m_1}(!f_{m_2}(!f_{m_3}(\dots !f_{m_{n_m}}())\dots))$	

Figure 4.9 – Emulation of the traditional top-down inside-out stage evaluation order in the integrated model; delayed stage evaluation is forced with quasi-quotes and inlining.

4.4 Discussion

There are certain tradeoffs involved in adopting the integrated metaprogramming model. On the one hand it offers the notion of a coherent metaprogram with its lexical scoping, shared state and sequential control flow. On the other hand, supporting these features introduces an explicit dependency across stage fragments that could restrict the potential for evaluating in a different way, such as arbitrary reordering or parallel evaluation. However, we consider the latter to be a general language issue rather than a metaprogramming model concern. In particular, as with normal programs, reordering or parallelism may be automated by optimizers and runtimes to improve performance, however, without mandating a paradigm shift from the original programming model of languages. Since we emphasize the common treatment of metaprograms and normal programs we consider all these issues on metaprograms to be uniformly addressed following the practices of normal programs.

Another tradeoff relates to the inherent programming complexity in managing and orchestrating separate stage fragments in order to behave meaningfully under their sequential control flow. Since our control flow links code segments together that are not close to each other in file scope, programmers may have to non-locally shift focus of attention to assimilate such behavior. Apparently, this is not an easy task and is something beyond the requirements of handling normal programs. However, its complexity is not the same as splitting an algorithm into disparate sections and keeping track of its behavior. In fact, we do not suggest or foresee that algorithms within stages are to be split for some reason into separate stage fragments. We consider that most dependencies between fragments will concern scope access to earlier state and behavior, with the stage control executing linearly across fragments from top to bottom of the main source file. Clearly, once stage fragments can be independent to each other, they are essentially executed atomically and are far easier to understand and control their behavior. While the latter is implemented with no extra complexity in our model, we believe such scenarios represent only a very small picture on what we expect metaprograms to do in the future.

Once a metaprogram grows at a point where it becomes difficult to manage its source code, runtime state and control flow, we can deploy refactoring, separate modules, abstraction techniques, or whatsoever, as with any normal program suffering from

similar issues. For instance, it is possible to make libraries with code composition functionality (AST manipulation, non-staged) that can be deployed by the various metaprogram stages. However, it is not possible to entirely decouple the functionality of a metaprogram from the original source file it actually affects. The actual source locations where the code generation occurs are determined by the inline tags placed directly within the affected source file. Thus, even if the entire metaprogram logic is placed in a separate module, the original source still needs staged code to load the module and call generator functions inside the appropriately placed inline directives.

Finally, a note related to type checking, as it enables an entire class of metaprogramming bugs to be caught early by the type checker. Since Delta is an untyped object-based language and the metaprogramming extension fully reuses the host language, it involves no type checking. However, the integrated metaprogramming model as such is orthogonal to the presence of a type system. Our proposition targets the composition and execution of stages and does not involve the type system, even if the hosting language would be typed.

4.5 Case Studies

We discuss metaprogram scenarios utilizing basic object-oriented features like encapsulation and state sharing. Such features may differ from what is typically met in the discussion of a metalanguage, but they are chosen on purpose to: (i) emphasize our point that metaprograms are more than atomic macro expressions; and (ii) highlight the importance of engineering stages like normal programs exploiting shared state and control flow among stage fragments. It should be noted that we do not argue the computation involved in such examples cannot somehow be expressed in existing multi-stage languages. Instead our focus is purely on the software engineering advantages of our model enabling typical programming patterns and techniques that are applied in normal programs. In general, most examples involve grouping of common functionality under generative objects that are shared across different stage fragments. Such objects are only setup initially and are freely deployed within different stage fragments to generate code. The latter avoids the tedious repetition of the setup sequence as required with atomic macros that lack state sharing.

4.5.1 Exception Handling

Exception handling [Goodenough] is known to be a global design issue that affects multiple system modules, mostly in an application-specific way. In this sense, it should be possible to select a specific exception handling policy for the entire system or apply different policies for different components of the system. Using typical object-oriented techniques, the only solution would be to abstract the desired exception handling policy within a function (or object method) and place a corresponding invocation to every applicable catch block. However, it does not avoid the boilerplate code required for declaring the handler and performing the function call, nor does it support arbitrary exception handling structures or context-dependent information.

In this context, we can use metafunctions to generate code for exception handling patterns. However, without shared state, metafunction invocations are separated and require explicit and tedious repetition of the pattern details. Moreover, if multiple exception handling patterns are available, it is not possible to parameterize their application, or even use binders, to form custom exception handling policies. Using our model, it is possible to maintain a collection of the available exception handling patterns and select the appropriate policy based on configuration parameters or normal control flow while requiring no changes at the call sites inside client code. This is illustrated in the following example.

```
&function Logging (stmts)
  { return << try { ~stmts; } catch e { log(e); } >>; }

&function CreateRetry (data) {      ← constructor for a custom retry policy
  return function (stmts) {        ← return a function implementing the code pattern
    return <<                      ← the returned function returns an AST
      for (i = 0; i < ~(data.attempts); ++i)
        try { ~stmts; break; }      ← try & break loop when successful
        catch e { Sleep(~(data.delay)); } ← catch & wait before retrying
      if (i == ~(data.attempts))    ← maximum attempts were tried?
        { ~(data.failure_stmts); } ← then give-up & invoke failure code
    >>;
  }
}

&ex = [                            ← compile-time structure for holding exception handling policies
  @policies : [], @active : "",
  method InstallPolicy (key, func) { @policies[key] = func; },
  method SetActivePolicy (policy) { @active = policy; },
  method Apply (code) { return @policies[@active](code); }
];
```

<code>&ex.InstallPolicy("LOG", Logging);</code>	\leftarrow install the logging policy
<code>&ex.InstallPolicy("RETRY", CreateRetry([</code>	\leftarrow create and install a retry policy
<code> @attempts : 5, @delay : 1000, @fail : <<post("FAIL")>></code>	
<code>]));</code>	
<code>&ex.SetActivePolicy("RETRY");</code>	\Rightarrow
<code>!(ex.Apply(<<f()>>));</code>	<pre> for (i = 0; i < 5; ++i) try { f(); break; } catch e { Sleep(1000); } if (i == 5) { post("FAIL"); } </pre>
<code>&ex.SetActivePolicy("LOG");</code>	\Rightarrow
<code>!(ex.Apply(<<g()>>));</code>	<code>try { g(); } catch e { log(e); }</code>
<code>!(ex.Apply(<<h()>>));</code>	<code>try { h(); } catch e { log(e); }</code>

As shown, we utilize the stage object *ex* to accommodate and compose typical exception handling policies. It is used in an object-oriented fashion to initially install a number of required policies, such as LOG and RETRY, and to generate the respective exception handling code by the invocation of the `Apply()` method. In this example, `Logging` is directly a policy metafunction, while `Retry` accepts parameters to produce the required policy metafunction (e.g. number of retries, delay between attempts and fallback code when all attempts fail). Such parameters are provided once, upon policy installation, and are not repeated per policy deployment. This relieves programmers from repeatedly supplying all required parameters and constructing all needed objects. Additionally, and most importantly, it allows a uniform invocation style, enabling different policies to be activated as required at an initial point, without inherent changes at the generation sites involving `!(ex.Apply(...));` directives. An extended version of this example as well as further exception handling patterns based on our model are discussed in section 7.2.

4.5.2 Design Patterns

Design patterns [Gamma] constitute generic reusable solutions to commonly recurring problems. They are not offered as reusable modules, but are recipes to apply a solution to a given problem in different situations. This means that in general, a pattern has to be implemented from scratch each time deployed, thus emphasizing design reuse as opposed to source code reuse.

We have examined the possibility of utilizing metaprogramming to support generating concrete pattern implementations, where applicable. In this context, the pattern skeleton is turned into composition of ASTs, the pattern instantiation options become composition arguments, the actual client code is supplied as AST arguments and the

pattern instantiation is handled by generative directives. To effectively accommodate such requirements, metaprograms require features beyond staged expressions.

With integrated metaprograms, programmers may apply practices like encapsulation, abstraction and separation of concerns, thus significantly improving the metaprogram development process. For example, it is possible to implement abstract pattern generators, have multiple such objects or even hierarchies of them available, and select the appropriate generator for a target context while preserving a uniform invocation style. This functionality is demonstrated in the following example that implements the *adapter* pattern. The pattern is implemented in two ways, using delegation and sub-classing, while its application may be parameterized with staging.

```
function Window(args) { ← runtime class that will be adapted
  return [
    method Draw() {...},
    method SetWholeScreen() {...},
    method Iconify() {...}
  ];
}

&function GetClassDef (target) {...} ← uses compiler state to find the target class

&function AdapterByDelegation() { ← creates an adapter object that uses delegation
  return [
    method adapt (spec) {
      local methods = nil; ← AST of adapted class methods, initially empty
      local class = GetClassDef(spec.original);

      foreach(local m, class.getMethods()) { ← iterate over class methods
        local name = m.GetName();
        local newName = spec.renames[name];
        if (not newName)
          newName = name; ← if no renaming use original name
        methods = << ← merge existing adapted methods with the current one
          ~methods,
          method ~newName (...) { @instance.~name(...); }
        >>;
      }

      return << ← create and return the adapted class using the adapted methods AST
        function ~(spec.adapted) (...) {
          return [
            @instance : ~(spec.original) (...),
            ~methods
          ];
        }
      >>;
    }
  ];
}
```



```

&function AdapterBySubclassing() { ← creates an adapter object that uses subclassing
  return [
    method adapt (spec) {
      local adaptedMethods = nil; ← AST of methods to be adapted, initially empty
      local class = GetClassDef(spec.original);
      foreach(local m, class.getMethods()) { ← iterate over class methods
        local name = m.GetName();
        local newName = spec.renames[name];
        if (newName) ← only check renamed methods, other are inherited by base class
          adaptedMethods = << ← merge adapted methods with the current one
            ~adaptedMethods,
            method ~newName (...) { self.~name(...); }
          >>;
      }
      return << ← the adapted class as a subclass that introduces the adapted methods
        function ~(spec.adapted)(...) {
          local base = ~(spec.original)(...); ← base class object
          local derived = [~adaptedMethods]; ← derived class object
          std::inherit(derived, base); ← derived object inherits from base
          return derived;
        }
      >>;
    }
  ];
}

AdapterFactory = [ ← Creating and populating a factory with adapter implementations
  @adapters : [],
  method Install (type, func) { @adapters[type] = func; },
  method New (type) { return @adapters[type](); }
];

AdapterFactory.Install("delegation", AdapterByDelegation);
AdapterFactory.Install("subclassing", AdapterBySubclassing);
adapterType = "delegation"; ← can also be read or computed dynamically
adapter = AdapterFactory.New(adapterType); ← create an adaptor object
windowAdapterData = [ ← compile-time data for the window adapter
  @original: <<Window>>, @adapted : <<WindowAdapter>>,
  @renames : [{"SetWholeScreen":"Maximize"}, {"Iconify":"Minimize"}]
];

! (adapter.adapt(
  windowAdapterData
));

function WindowAdapter(...) {
  return [
    @instance : Window(...),
    method Draw(...) { @instance.Draw(...); },
    method Maximize(...)
      { @instance.SetWholeScreen (...); },
    method Minimize(...) { @instance.Iconify(...); }
  ];
}

adapter = AdapterFactory.New("subclassing"); ← create new adapter object
windowAdapterData.adapted = <<WindowAdapter2>>; ← change adaptation data

! (adapter.adapt(
  windowAdapterData
));

function WindowAdapter2(...) {
  local base = Window(...);
  local derived = [
    method Maximize(...)
      { self.SetWholeScreen(...); },
    method Minimize(...) { self.Iconify(...); }
  ];
  std::inherit(derived, base);
  return derived;
}

```

Such generator objects can also abstract implementation details of the classes they produce, with such details specified only upon creation. For instance, consider a *Singleton* class that may adopt different invocation styles (e.g. static functions or static instance and methods), that may even be declared within a namespace, thus requiring extra syntax in its usage. Implementing such a code generation scheme in a typical multi-stage language requires repeating the generated class details at every location, something painful (consider that such details are syntactically verbose due to quasi-quotes) and error-prone. Similarly, updating or replacing the implementation would require manually locating all affected sites and applying individually the required changes. Below we show an example for the definition and usages of a *MemoryManager* singleton class implemented in a hypothetical meta-C++ language adopting our staging tags and integrated metaprogramming model. Notice that in this particular example, there are no name conflicts across global and local declarations, so for simplicity we do not use any *define* tags but only *execute* tags.

```
&AST* impl = <<                                     ← basic MemoryManager class implementation
    void Initialize () {...}
    void Cleanup () {...}
    void* Allocate (n) {...}
    void Deallocate (void* var) {...}
>>;
&class MemoryManagerGenerator {                       ← MemoryManagerGenerator interface
protected:
    AST* namespace;  ← namespace in which the target class will reside in (may be null)
    AST* GetClass() {← if target class is within a namespace, use a fully qualified name
        AST* result = <<MemoryManager>>;
        if (namespace) result = << ~(namespace)::~result >>;
        return result;
    }
public:                                                 ← methods to be implemented by concrete generator subclasses
    virtual AST* GetDef() = 0;
    virtual AST* GetInit() = 0;
    virtual AST* GetCleanup() = 0;
    virtual AST* Allocate (AST* n) = 0;
    virtual AST* Deallocate (AST* var) = 0;
    MemoryManagerGenerator (AST* ns = 0) : namespace(ns) {}
};
&class StaticFuncGenerator : public MemoryManagerGenerator{
private: AST* MakeAllFunctionsStatic(AST* funcList) {...}
public:
    AST* GetDef() const { ← generate singleton class using static functions
        AST* staticImpl = MakeAllFunctionsStatic(impl);
        ← modify base MemoryManager implementation making all of its functions static
        AST* result = <<class MemoryManager { public: ~staticImpl; };>>;
        if (namespace) ← wrap the class within the given interface, if any
            result = << namespace ~namespace { ~result; } >>;
        return result;
    }
    AST* GetInit() { return <<~(GetClass())::Initialize()>>; }
```

```

AST* GetCleanup(){ return <<~(GetClass())::Cleanup()>>; }
AST* Allocate (AST* n){ return <<~(GetClass())::Allocate(~n)>>; }
AST* Deallocate(AST* v) {return <<~(GetClass())::Deallocate(~v)>>;}
StaticFuncGenerator(AST* ns = 0) : MemoryManagerGenerator(ns) {}
};

&class StaticInstanceGenerator : public MemoryManagerGenerator {
private:
    AST* GetInstance() const { return <<~(GetClass())::Instance()>>; }
public:
    AST* GetDef() const {
        ← generate singleton class using a static instance
        AST* result = <<
            class MemoryManager {
            public:
                static Instance() {
                    static MemoryManager instance;
                    return instance;
                }
                ~impl; ← insert methods from base MemoryManager implementation
            };
        >>;
        if (namespace) ← wrap the class within the given interface, if any
            result = << namespace ~namespace { ~result; } >>;
        return result;
    }
    ...all following methods use GetInstance to generate code for accessing the static instance...
    AST* GetInit(){ return <<~(GetInstance())->Initialize()>>; }
    AST* GetCleanup(){ return <<~(GetInstance())->Cleanup()>>; }
    AST* Allocate (AST* n) {return <<~(GetInstance())->Allocate(~n)>>;}
    AST* Deallocate(AST* v){return <<~(GetInstance())->Deallocate(~v)>>;}
    StaticInstanceGenerator(AST* ns = 0) : MemoryManagerGenerator(ns){}
};

&MemoryManagerGenerator* mm = new StaticFuncGenerator(<<Memory>>);
    ← create a concrete code generator object and use it through the base class API

```

<pre> ! (mm->GetDef()); </pre>		<pre> namespace Memory { class MemoryManager { public: static void Initialize () {...} static void Cleanup () {...} static void* Allocate (n) {...} static void Deallocate(void* var){...} }; } </pre>
<pre> ...other global definitions... int main() { ! (mm->GetInit()); ...other normal program initializations... void* x = ! (mm->Allocate(<<10>>)); ...normal code using variable x ... ! (mm->Deallocate(<<x>>)); ...other normal program code... ! (mm->GetCleanup()); ...other normal program cleanups... return 0; } </pre>		<pre> ...other global definitions... int main() { Memory::MemoryManager::Initialize(); ...other normal program initializations... void* x = Memory::MemoryManager::Allocate(10); ...normal code using variable x ... Memory::MemoryManager::Deallocate(x); ...other normal program code... Memory::MemoryManager::Cleanup(); ...other normal program cleanups... return 0; } </pre>

```

&delete mm; ← dispose the compile-time generator object

```

As shown, the invocation details are specified only once for each case and are abstracted through the mm code generator object, allowing the definition and

deployment code to be automatically produced without requiring any extra information. The latter allows updating the generation parameters, possibly affecting names or calling styles, without having to change all client uses of the generated class. Also, in this example, the ordering of the inline directives is important. In particular, the definitions regarding the memory manager object should be generated before its actual deployment, thus `!(mm->GetDef())` is put first. In the same sense, the initialization statements of a memory manager should be generated before any memory allocation calls, thus `!(mm->GetInit())` is put next. Then, any cleanup actions usually take place the end of the program so the `!(mm->GetCleanup())` is put last. A working example of similar MemoryManager functionality implemented in Delta is available in [Lilis13], while further design pattern examples based on metaprogramming are discussed in section 7.1.

4.5.3 Design By Contract

Design by Contract (DbyC) [Meyer91] is a popular method towards self-checking code improving software reliability. It proposes contracts, constituting computable agreements between clients and suppliers. Clients have to respect method *preconditions* prior to invocation while suppliers guarantee that the associated *postconditions* will be satisfied once the invocation completes. Failure to satisfy the promised obligations, on either the client or the supplier side, constitutes a contract violation that will most likely result into an error, typically conveyed as an exception.

In this context, it is possible to use metaprogramming to automatically generate contract verification code. This applies both for the supplier class, whose methods can be enriched with precondition and postcondition checking that raise exceptions upon contract failures, and the class clients, whose invocations can be automatically protected with try-catch blocks. However, the definition of the supplier class is separated from the client invocations, meaning that the applications of the code transformations are also typically separated. This means that if the transformation logic is not known a priori, i.e. it relies on some prior compile-time computation, it is not possible to match the generated class definition with a corresponding generation of the class invocations. Even if the transformation logic is predefined, its applications are still separated so they may be applied partly, meaning it is possible to end up with a supplier class that uses DbyC and client invocations that do not or vice versa. In the

first case any thrown supplier exception will never be handled by clients, while in the second case client invocations will contain irrelevant exception handling code since the supplier class may not throw any contract exceptions.

This problem can be solved with the state sharing and typical control flow offered by integrated metaprograms. Any transformation to be applied on the supplier class can be stored along with the corresponding transformation required for its usage and be available in the following stage calls that will generate the client invocations, taking into account the transformations performed on the class definition. The following code highlights this functionality, by introducing a single object that can be used to transform both the class definition (through the `std::context` function discussed earlier) and usages. In particular, the transformer object `t` contains all relevant transformation information and could be used to handle any number of classes along with their usages. Additionally, notice that the inlining code that uses the transformer object is completely unaware of the actual transformation being applied; this information is properly encapsulated within the transformer object.

```
&function DbyC() {
  return [
    method supplier(class) {
      foreach (local m, class.getMethods()) {
        local pre_id = "pre_" + m.getName();
        if (class.hasMethod(pre_id))
          m.body.push_front(
            <<
            if (not self[~pre_id]())
              throw [
                @class: "ContractException",
                @type : "Precondition",
                @classId : ~(class.getName()),
                @method: ~(m.getName())
              ];
            >>
          );
      }
      ...similar logic to add postcondition checking code at the method end here...
    }
    return nil;
  ],
  method client(invocation_stmts) {
    return <<
      try { ~invocation_stmts; }
      trap ContractException { log(ContractException); }
    >>;
  }
];
}
```

```

&t = DbyC();           ← compile-time transformer object
function Stack() {
  return [
    method empty    {...},
    method pre_pop  {...},
    method pop       {...},
  ];
}

!( t.supplier(
  std::context("class")
));

function Stack() {
  return [
    method empty    {...},
    method pre_pop  {...},
    method pop       {
      if (not self["pre_pop"]())
        throw [
          @class : "ContractException",
          @type  : "Precondition",
          @classId: "Stack",
          @method : "pop"
        ];
      ... original body of pop method follows here...
    },
  ];
}

st = Stack();
!(t.client(<<st.pop()>>));

st = Stack();
try { st.pop(); }
catch ContractException
{ log(ContractException); }

```

The same approach can be extended to handle additional transformations that affect both the definition and usage of a given class. For instance, if we wanted the Stack class of the previous example to also be transformed as a singleton class, we could define a corresponding transformer meta-function and then combine the two to create a new transformer object that can be used without affecting any of the code generating invocations.

```

&function Singleton() {           ← Singleton transformer
  return [
    method supplier(class) { ...turn class into a singleton... },
    method client(code) { ...generate invocation style for singleton client... }
  ];
}

&function ListTransformer(list) { ← combination of multiple transformers
  return [
    @list : list,
    method supplier(class) {
      local retval = nil;
      foreach(local t, @list)           ← combine all class definitions
        retval = <<~retval, ~(t.supplier(class))>>;
      return retval;
    },
    method client(code) {               ← combine all client invocation uses
      foreach(local t, @list)
        code = t.client(code);
      return code;
    }
  ];
}

&t = ListTransformer(list(DbyC(), Singleton()));
...the rest of the code remains exactly as it is ...

```

Chapter 5

Tool Extensions

"Man is a tool-using animal. Without tools he is nothing, with tools he is all."

- Thomas Carlyle

Next to programming languages, Integrated Development Environments (IDEs) and the tools they provide play the most critical role in the software development process. In this sense, an Integrated Metaprogramming System requires, apart from the extension in the language, a set of tool extensions able to support metaprogramming in a similar way current tools support normal programming. Such extensions include the integration of metaprograms and their outcomes in the workspace manager allowing code review and source browsing, a build system aware of stage metaprograms and custom build properties they may involve, a facility for proper reporting of compile errors generated by metaprograms and finally full-fledged source-level debugging for both metaprograms (during compilation) as well as for generated final programs (during runtime). We continue elaborating on the integration of stages in the IDE of the Delta language, *Sparrow* [Savidis07], and discussing the necessary interactions between the compiler and the various IDE subsystems.

5.1 Workspace Manager

When using metaprogramming the code available in the source file and the code that is finally compiled to byte-code may be significantly different. This, of course, is due to the code transformations defined in the metaprogram itself, but results in executing code that was never part of the original program and that the programmer may be unable to see or understand. This can be a major drawback especially in cases where the final program does not behave as expected and the programmer is unable to determine why. The reason could be a faulty implementation of the metaprogram, a misuse in its application in the current context or even some error in the normal program code; however the programmer only sees the erroneous behavior with no additional feedback about its origin.

To facilitate programmers in such situations, we extract and store stage metaprogram source code as separate read-only files that are incorporated in the workspace manager of the IDE associated with their respective main program. Clearly, this allows for better reviewing and understanding, compared to studying a metaprogram across disparate text fragments as embedded in the main program. Additionally, we also store the modified versions of the main program, after every stage evaluation showing clearly the staged transformations introduced over the original program. Essentially, this means that for a compilation involving n stages, there will be a total of $2 \cdot n$ source files generated, that are collectively referred to as *compilation stage sources*. All such source files are produced during the compilation process and are propagated to the workspace manager for inclusion in its respective folders and structures. Once incorporated in the workspace, besides reviewing and browsing, they are also used for reporting compile errors (see section 5.3), setting breakpoints and performing source-level debugging (see section 5.4).

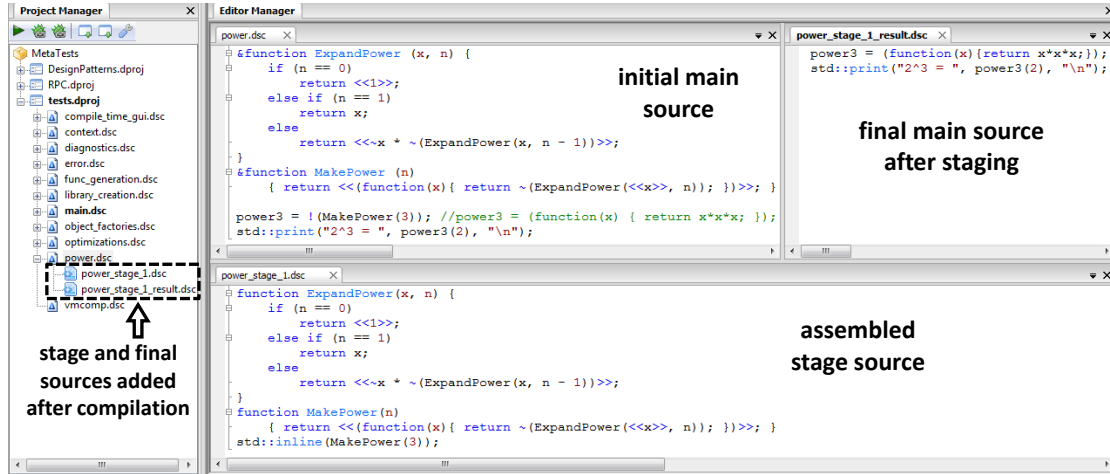


Figure 5.1 – Reviewing the compilation sources in Sparrow: Project manager view (left), initial main source (middle), assembled stage source (bottom), final main source after staging (middle right).

Extracting the source code for compilation stage sources requires utilizing the respective ASTs that are available during the compilation process and producing a textual representation of their code, a process known as unparsing. Source code for stage metaprograms is generated based on the assembled stage program AST (Figure 5.2, denoted as *AST of stage i*), while source code for main program transformations is generated based on the main program AST versions that are updated as a result of the corresponding stage execution (Figure 5.2, denoted as *Intermediate Main AST after stage i*). The generated source code is then stored using some naming

convention, for example adding a suffix along with the current stage number (Figure 5.2, denoted as *stage_i_source* and *stage_i_result_source* for stage metaprograms and their results). In particular, the final program being compiled into executable code is actually the result of the last stage, so it will also be available as the last generated source (Figure 5.2).

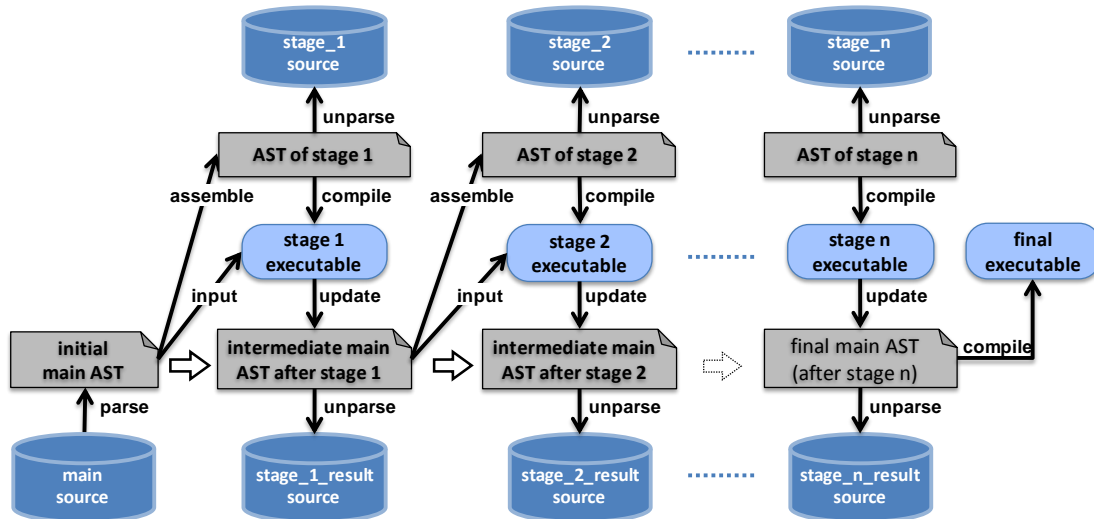


Figure 5.2 – Storing the source code of all stage metaprograms and their outputs (main program transformations).

The compilation stage sources are meant to be shown to programmers, so naturally they have to be as readable as possible. This means that any generated code segments should span across multiple lines and be properly indented. Nevertheless, both stages and main program transformations may contain code originating directly from the main source. User written code should clearly be preferred over automatically unparsed code (different indentation, empty lines, comments, etc.), so any such code segments should maintain its original form. To support this efficiently, AST nodes contain their starting and ending character positions in the original source allowing the retrieval of their text segments. This way, the unparsing algorithm can combine original and generated text segments to produce a visually appropriate source file.

Once such a source file has been generated during compilation it is sent to the IDE to be incorporated into the workspace. This interaction and the communication it involves rely on the way the compiler is invoked by the IDE. In case the compiler is available as an IDE service invoked during the build process, it is possible to directly provide callbacks for specific events like compilation errors or generation of

compilation stages source. If it is implemented as a separate executable spawned by the IDE, the communication channel between them is typically a memory pipe using standard text input and output facilities. This requires establishing a protocol for communicating the compiler events to the IDE using some text representation. For example, to notify the IDE about the existence of the compilation stage sources, the compiler may use a special message containing resource identifiers for them (e.g. file paths). Early versions of our metaprogramming systems actually adopted this method since it required only minimal extensions on the existing message handling infrastructure. One last alternative is for the interaction to fully rely network based communication. In this case, the IDE acts as a server for receiving compilation stage sources and supplies the host and port information as parameters in the compiler invocation. Once launched, the compiler will use these parameters to establish a connection with the IDE and use it for supplying it with any compilation stage sources generated during the compilation process (see section 5.2). This is the approach currently adopted in our metaprogramming system. The reason for changing the original approach was that the compilation stage sources required additional metadata associated with them in order to properly support compile-time debugging of stages. This will be discussed in more detail later in section 5.4.2).

5.2 Build System

To treat metaprograms equally to normal programs we need to support them with typical build and deployment tools. While a metaprogram resides in a source file along with a normal program, it may use external libraries or compilation options that are entirely different from those used by the normal program and clearly, programmers should be allowed to specify them despite the fact that metaprogram sources are not available prior to compilation.

Towards this direction, we extend the deployment options associated with normal sources to accommodate information about stages. In particular, we support specifying custom compilation options (e.g. additional external libraries, compiler options, deployment options, etc.) for specific stages while also providing default options applicable for all other stages. The default options are actually very useful when the number of stages cannot be statically determined, thus disabling the association of compilation options with specific stage numbers.

With the stage compilation options specified directly within the normal program deployment options, it would be possible to provide them as arguments in the compiler invocation. Then the compiler could utilize the supplied options, handling the assembly and compilation of the stage metaprograms internally, without requiring any additional interaction. Nevertheless, this deployment model does not support metaprograms to have any dependencies; before the compiler invocation they are not available while during it they cannot be resolved without external information. The latter essentially dictates that in order to support this functionality the compiler should directly interoperate with the build system. In this context, we utilize the ability to incorporate compilation stage sources directly in the workspace manager, generate their compilation options based on the stage compilation options of the main program by matching their stage numbers and allow the compiler send build request for them directly to the build system. The entire process, outlined under Figure 5.3, involves the following steps:

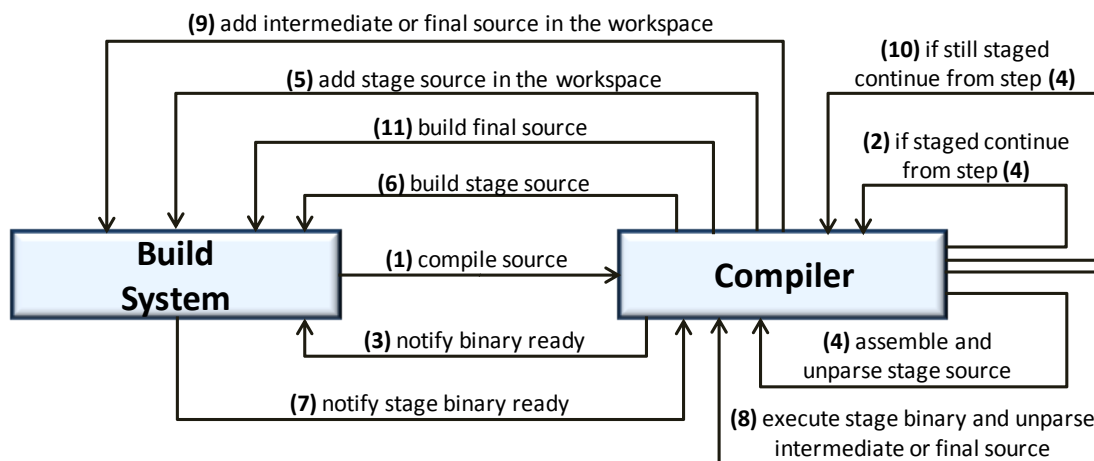


Figure 5.3 – Build system and compiler interaction sequence diagram regarding metaprograms.

1. When a source is to be compiled, the build system resolves and builds its dependencies, finally invoking the compiler on the actual source, while also waiting possible further requests from it.
2. If the source contains stages the process continues from step 4.
3. Otherwise, the source is directly compiled, the build system is notified that the binary is ready, and the compilation process ends here.
4. The stage source, i.e. an integrated metaprogram, is composed.

5. It is submitted to the build system for propagation to the workspace manager. Any compilation options for this source are derived from the stage compilation options of the originating main source.
6. The build system is asked to build the submitted stage source.
7. Once ready, the build system responds that the stage binary is ready (else, error reporting is involved).
8. The metaprogram is run, modifying the current AST, and the transformed main source is produced by unparsing the AST.
9. It is submitted to the build system for propagation to the workspace manager. As this is a new version of the main program, it inherits its compilation options directly from it.
10. If it is still staged then we continue from step 4.
11. Otherwise, it is the final source and the build system is asked to build it.

Effectively, the compiler becomes a client of the build system, capable to recursively involve the build system back in the loop with additional build requests for stages.

For subsequent build requests we also have to consider when a source should be rebuilt or when it should be considered up-to-date, maintaining its binary without involving any compilation. Since stages originate from code within the main source, modifications in its code may result into different staged code and eventually a different final program. Stage sources are automatically generated and thus semantically read-only; however they may still be outdated if one of their dependencies has changed. In general, this means different stage execution, resulting to a possibly changed intermediate program, and consequently, again, a different final program. To properly support such cases, sources that contain stages should be handled specially and thus the entire build process has to be extended to become staging-aware, as illustrated in the flowchart of Figure 5.4. In particular, if the target source contains stage sources (outcome of a previous build session), the recursive build of dependencies is omitted as the main source is bound to change due to metaprogramming. Instead the build system checks if the target source is up-to-date or it has changed since the last build (with respect to its previously produced binary). If it has changed, any compilation stages sources will also change, so they are removed from the workspace and the target source is sent for compilation. Any dependencies within the main source that are not yet processed will remain intact

during the program transformations (new dependencies may be introduced via meta-programming, but the original ones cannot be removed), so they will be available in the final source, at which point they will be recursively built before compiling the final program source. If the main source is up-to-date, we further check if its associated stage sources are up-to-date which practically means checking whether any of their dependencies have changed. If this is the case, the resulting final program is also outdated, so any compilation stage sources are again removed from the workspace and the target source is sent for compilation. Otherwise, all stage sources are up-to-date and consequently the generated final program remains the same. This way, the only additional step required is to normally build the final source. If any of its dependencies have changed, they will be recursively built, followed by a compilation of the final source, while if they are unchanged, the final source itself is unchanged so the build process will yield up-to-date.

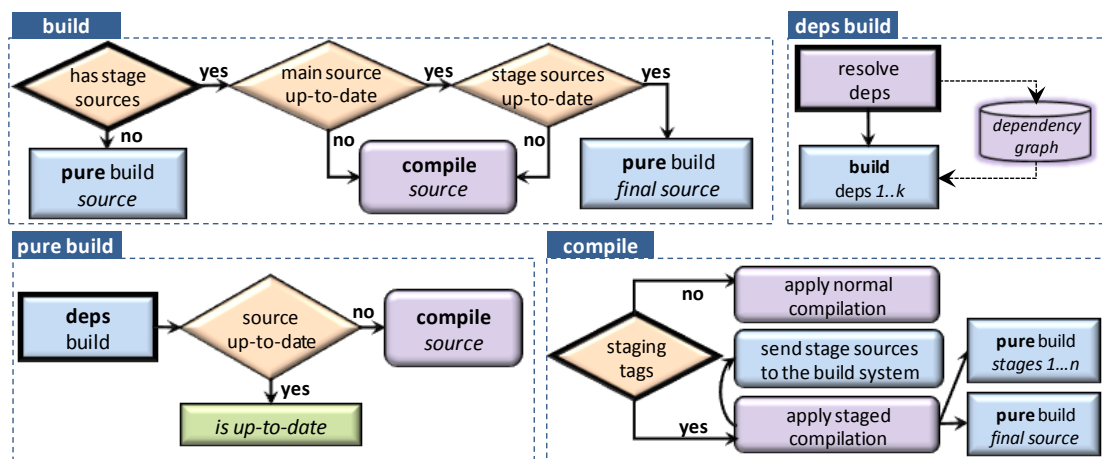


Figure 5.4 – Control flow for the staging-aware build process; starting process is ‘build’ (top-left).

It is worth noting that the first compiler invocation on a source file with stages performs no actual translation. Instead, its purpose is to produce the stage sources, supply them to the build system, and when done, execute their binary to apply their transformations on the main program. Once done with stages, the compiler will eventually submit the final source of the transformed main to the build system.

5.3 Compile-Error Reporting

As with normal programs, when writing metaprograms errors are bound to happen, so it is important to have the proper tools to understand their origin and finally resolve them. Compilation errors are generally considered easy to resolve as compilers can

identify exactly where something went wrong and why. However, in the context of metaprogramming, compilation errors tend to be much harder to resolve; metaprogram compilation may involve code that was never part of the original program and thus the traditional error report may no longer reflect code that the programmer can see and understand. A first step towards the solution is to utilize the compilation stage sources that are already incorporated in the workspace in order to provide the missing source information. Nevertheless, the source code of a metaprogram may be the result of a nested metaprogram and thus an error reported in the former may in fact be caused by the latter. Essentially, compile-error reports should encompass additional information regarding the context in which a metaprogram was assembled so as to help programmers identify the real cause of an error. We continue elaborating on the required information and introduce the notion of a compile-error chain across the compilation stage sources.

5.3.1 Compile-Error Chain across Stages and their Outputs

Both stages and their outputs, i.e. the main program transformations (including the final program) are derived from previous intermediate versions of the main program and ultimately from the original main program. As such, even code segments that were never part of the original program can be traced back to some source location of the original main program. In this sense, a meaningful and precise report for compilation errors occurring either in stages or the final program should encompass the entire code generation trajectory that led to the generation of the erroneous code segment. From the perspective of the error report, the latter allows creating an *error chain* across all involved source files (both stages and main program transformations) and combine this information into a descriptive message.

To support such functionality, each AST node is enriched with information about its origin, thus creating a list of associated source references. The source references for each node are created using the following rules:

1. Nodes created by the initial source parsing have no source reference.
2. When assembling nodes for a compilation stage, a source reference is created, pointing to the current source location of the node present in the main AST.
3. When updating the main AST, the source locations of the modified nodes are mapped to the latest stage source, creating the corresponding source reference.

Rules 1 and 3 along with the fact that the main AST can be modified only through the execution of the compilation stages guarantee that the main AST nodes will always either be a part of the original source or be generated by some previous stage and have a source reference to it. Furthermore, rule 2 and the fact that compilation stages are created using only nodes from the main AST guarantee the same property for all compilation stages as well. This means that any AST being compiled, either for some compilation stage or the final program, will incorporate for each of its nodes the entire trajectory of the compilation stages involved in their generation. Figure 5.5 provides a sample visualization of this information upon the occurrence of an error.

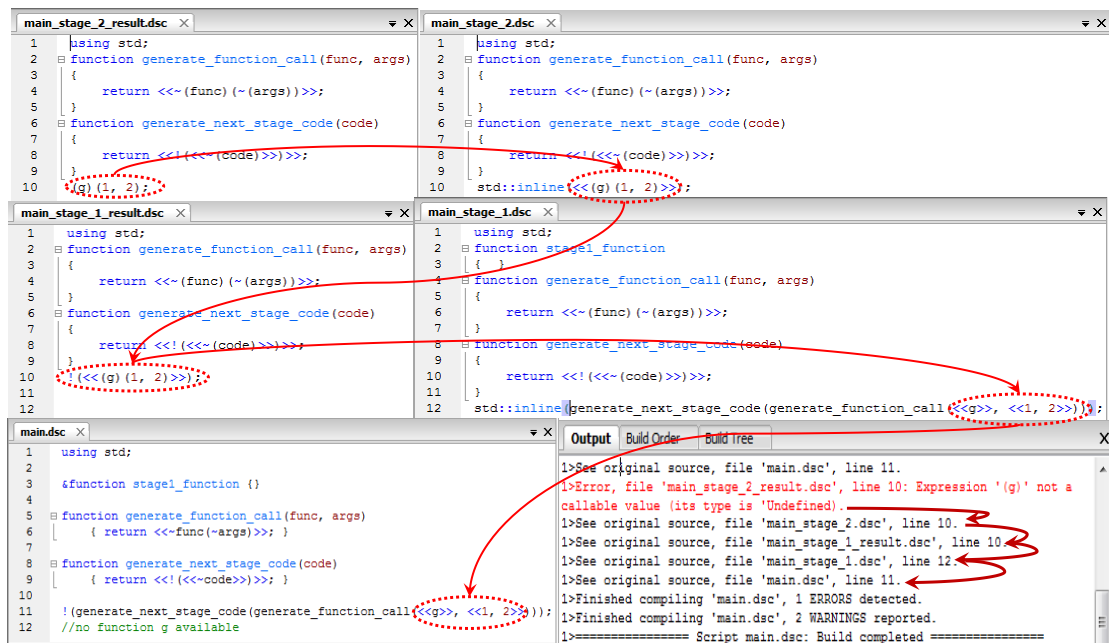


Figure 5.5 – Precise error reporting for compilation stages using the chain of generated sources.

We should note that our approach focuses on providing the source locations involved in a compile error but does not affect the error message itself. Since the same compiler executable is used for both normal programs and compilation stages, the same messages are naturally reported upon errors regardless of their origin. We believe that such error messages are chosen by the compiler to provide all relevant information based on the error context and are not related to the meta-compilation process. By providing the complete error chain across all stages and outputs, we provide the *missing information context* required to fully understand the error report.

For instance, the error reported in Figure 5.5 concerns an undeclared symbol called as a function. Looking only at the original source (bottom left) there is no evidence of

such a function call so the error message makes no sense; however looking at the stage result that actually caused the error (top left) we can spot the erroneous function call and thus understand the message shown in the error report.

5.3.2 Study for Runtime Metaprogramming

The above discussion focuses on a system offering compile-time metaprogramming; however error reporting is equally important for runtime metaprogramming. Code assembled at runtime may also generate errors during its translation, and thus require tracking down their origin to be resolved. In this context, we discuss how a methodology similar to that discussed previously could be deployed in languages supporting runtime metaprogramming, using existing features or possible extensions.

Let's consider multi-stage languages that generate code during runtime based on staging annotations. While it may be possible to ensure the type-correctness of a generated program based on the type-correctness of its generator (for instance this is the case in *MetaML*), there may also be semantic errors that cannot be reported before translating the code at runtime. For such cases, staging annotations should keep track of their locations and combine this information upon AST creations and combinations or insertions into the main program. For a single stage this would resemble the approach used by Converge for compile-time error reporting. For instance, consider the following example written in *Mint* [Westbrook], a multi-stage extension of Java.

```
Code<Void> code = <|{break;}|>;
for (int i = 0; i < 10; ++i) code.run();
code.run();
```

Essentially, the above code creates a delayed computation for a break statement and runs it both inside and outside of a loop. When the computation is run inside the loop it should normally execute the break statement and exit the loop. However, when it is run outside of a loop it should produce an appropriate error message referring to the origin of the erroneous computation. In this sense, the code object `<|{break;}|>` should maintain the line information of its origin, possibly combine it with line information from any involved escapes (in this case we have none), and finally use it during the execution of the run operator to provide an error message similar to the following: “In line 3, run introduces a ‘break’ outside of a loop. See original delayed computation at line 1.”.

However, if we have multiple stages or if the AST generation involves combining multiple quasi-quotes present in the original source, reporting just the original source locations of the code segments that generated the erroneous AST would probably be insufficient as any context of the final AST being translated is not available. In this case, we could unparse each of the involved ASTs to generate source segments (as separate files or parts of a file containing all the relevant information) that can then be used for linking the source references of the error message, thus providing the full context of any AST combination and insertion in the entire code generation chain.

In the context of runtime metaprogramming through reflection, the source code to be generated is just typical program data (e.g. dynamic text) and has no special source or line information associated with it. As such, the only way to provide an improved error reporting mechanism in this case, would be to manually insert any source and line information for the generated code along with any potential references to other source locations, if of course such a construct is supported by the language. For example, in C# it is possible to utilize the `#line` directive along with `System.Diagnostics.StackTrace` instances to allow instrumenting the dynamic source code in a way that it reports any of its compilation errors directly on the original source lines that generated the error. This functionality is depicted under Figure 5.6.

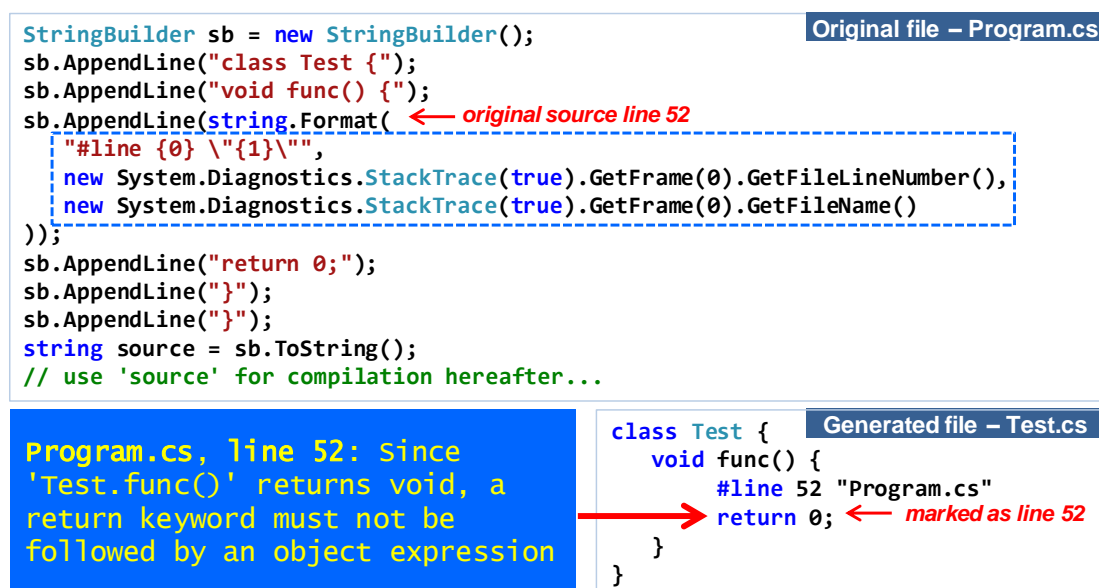


Figure 5.6 – Instrumenting `#line` directive for better error reporting in C#: The compilation error occurs in the generated file but is reported in the original file.

If the reflection API provides support for compiling code based on some AST value, it would be possible to enrich the AST structure with custom source file, line, or source reference information and have the programmer explicitly write code that sets this information in the nodes of the AST to be translated. This way, and considering that the compiler is also extended to utilize such AST information, it would be possible to generate a more detailed error report relating the generated code to the original program instructions producing it and allowing a client programmer identify the cause of an error more easily.

5.4 Source-Level Debugging

Every compilation stage is instantiated by the execution of the respective stage metaprogram. As such, it should be subject to typical source-level debugging even though it is executed during compilation. *Sparrow* provides such functionality supporting typical debugging facilities such as expression evaluation, watches, call stack, breakpoints and tracing. Figure 5.7 shows a compile-time debugging session highlighting the following points:

1. Breakpoints are initially set within a meta-function in the original source file.
2. The source file is built with debugging enabled. This launches the compiler for the build and attaches the debugger to it for any staged program execution.
3. During compilation, the IDE is notified about any compilation stage sources.
4. Stage sources are added in the workspace associated with the source being built.
5. A breakpoint is hit, so execution is stopped at its location.
6. The source corresponding to the breakpoint hit is opened within the editor to allow further debugging operations such as tracing, variable inspection, etc.
7. The breakpoints in the generated stage source (including the one hit) were automatically generated based on the breakpoints set in the original source file.
8. The execution call stack is available for navigation across active function calls.
9. It is possible to inspect variables containing code segments as AST values.

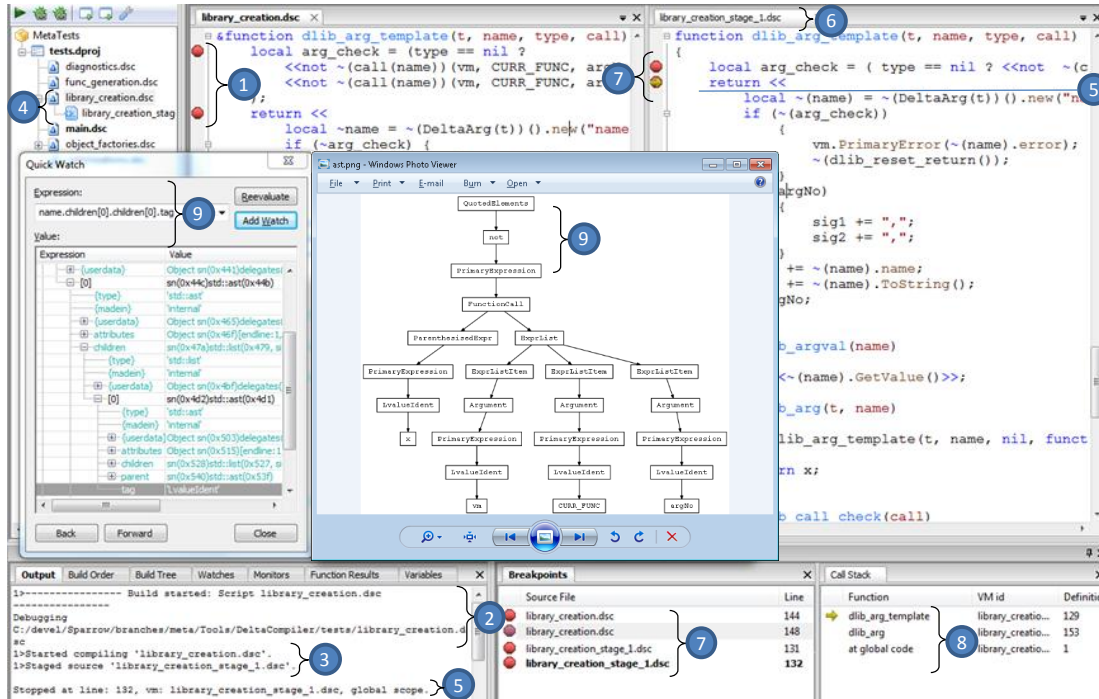


Figure 5.7 – A compile-time debugging session in Sparrow; highlighted items 1-9 are discussed in text.

Offering this functionality requires addressing three main issues: (i) initiating a compile-time debug session while utilizing the previously discussed generated stage sources for source information; (ii) introducing breakpoints for the stage code both before and during the debug session; and (iii) enabling inspection of AST values. We continue by detailing how each of these issues has been addressed within our system.

5.4.1 Translation-Time Debug Session

Stage programs are essentially normal programs, so it is possible to extend the standard language and IDE infrastructure to support them with typical source level debugging.

Generally, a debugging infrastructure is split in the backend, attached to the executing program (i.e. the debuggee) and providing an appropriate query protocol, and the frontend, offering user interaction and internally communicating with the backend. The backend is typically incorporated into the language runtime, and the frontend is usually part of the IDE. In our case, the debuggee is the compiler executable. Since the compiler is responsible to run stages, besides the language runtime, it must be linked with the backend as well (Figure 5.8).

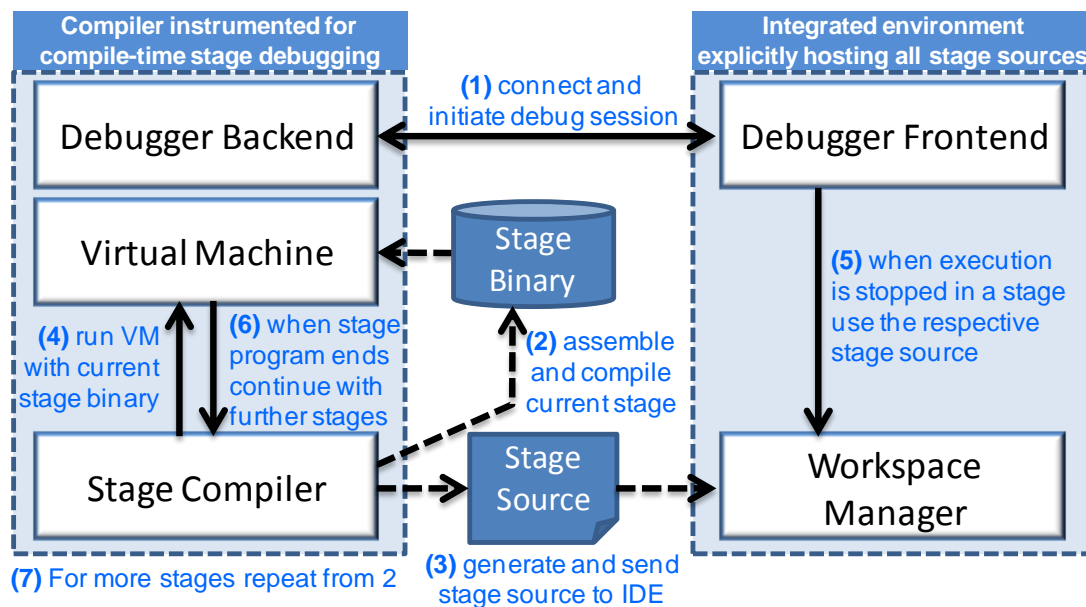


Figure 5.8 – Interaction between the compiler and integrated development environment for supporting compile-time source-level stage debugging.

To allow specifying that any stages executed during the compilation should be debugged the IDE should offer an explicit user option. Now, consider that stage debugging is enabled when building a source file that contains stages. This will launch the compiler, which in effect activates the debugger backend and connects to the debugger frontend (Figure 5.8, step 1). Then, the debugging session operates as with any other program. When a stage source is generated, the IDE is notified about its existence and incorporates it into the project management thus allowing its source to be used for debugging purposes. The stage is then translated to binary form and executed by the virtual machine (Figure 5.8, step 4). The latter is actually the only step relevant to the debugging process with the backend being attached to the stage binary execution. Apart from that, the stage build process and the transition to the next stage when the current terminates are totally transparent to the backend. In fact, this allows for the entire staging process to be debugged in a seamless way; stepping from the last instruction of one stage will cause the execution to pause at the first instruction of the next stage without requiring a separate debug session or any additional extensions.

From the IDE perspective, another requirement for proper compile-time debugging is to properly orchestrate any facilities previously targeted only for build or debug sessions. Essentially, IDEs provide different tools during build sessions (e.g. error

messages, build output, etc.) and debug sessions (e.g. call stack, watches, threads, processes, loaded modules, etc.), while usually applying different visual configurations for each activity. Compile-time debugging involves both a build and a debug session, so the provided facilities should be combined in a way that maintains a familiar working environment.

5.4.2 Supporting Stage Breakpoints

Once a stage source has been generated and incorporated into the IDE workspace, it is possible to normally add or remove breakpoint for the code it contains, however, that only happens during a meta-compilation round. Prior to that there are no stage sources and no way to associate breakpoints with their execution. The only information available is limited to any breakpoints associated with the main source being compiled. However, it is possible to use these breakpoints to automatically generate breakpoints for the involved stage sources.

The stage AST is composed of main AST nodes and, as previously discussed, main AST nodes are either part of the original source or recursively generated by some previous stage. Effectively, this means that each stage node originates from one or more nodes of the original AST. Viewing this from a different perspective, the original AST nodes can be associated with the stage nodes they generate, and the same applies for their line information. To achieve this, we extend the unparsing process discussed earlier to associate each node line of the AST being traversed to the current line of the source being generated, taking into account the lines introduced by the unparsing implementation (Figure 5.9). Each AST node generates two (not necessarily distinct) line associations, one upon entering and one upon exiting the traversal. The result of the traversal is a list with line associations that can be used to transform breakpoints placed in the original source into stage breakpoints.

As shown in Figure 5.9, the generated line associations are not necessarily one-to-one; effectively this means that a single original source breakpoint may end up generating multiple stage source breakpoints (e.g. a single-line expression that generates a multi-line function) while multiple source breakpoints may end up generating the same stage source breakpoint (e.g. a complex multi-line expression that generates a single line of code). However, this is actually the functionality that a programmer would expect supposing that any code modifications occur directly at the line they appear in

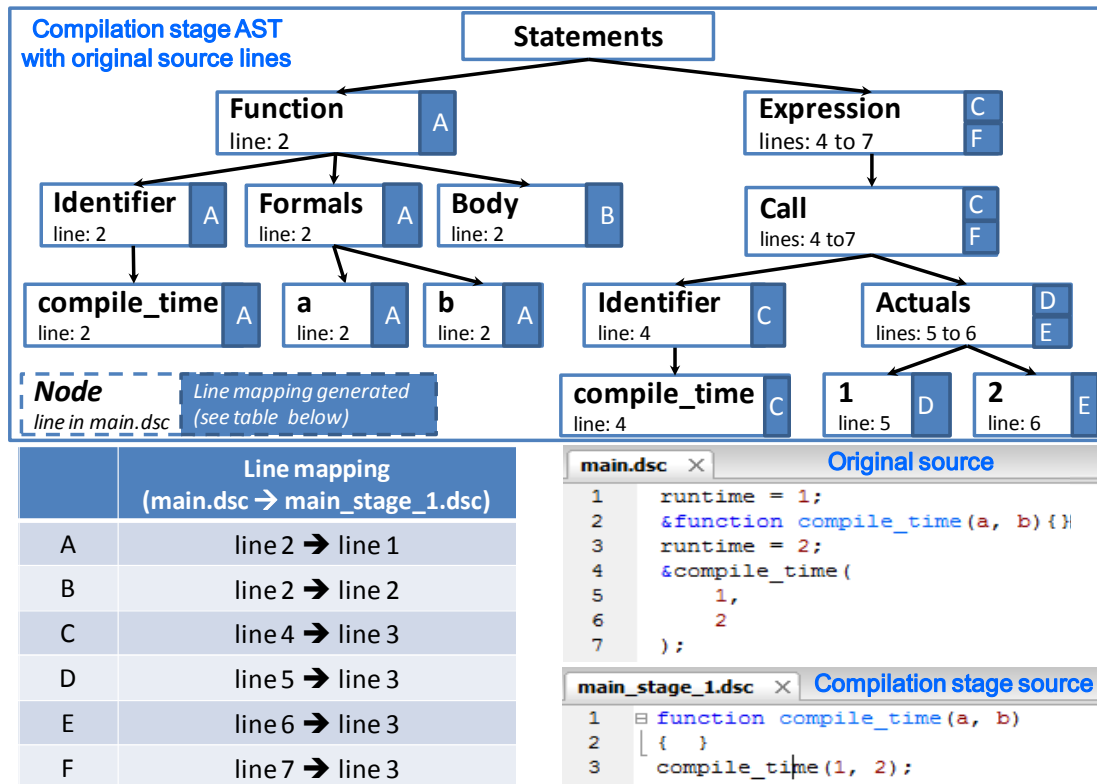


Figure 5.9 – Extracting line mappings for a compilation stage: The assembled stage AST (top), the original source and the compilation stage source (bottom right) and the line mappings generated by each AST node (next to each of the AST nodes, referring to elements of the bottom left table).

within the original source. For instance, an expression generating a function can be seen as substituting itself with a single (probably long) line containing the function definition. A breakpoint set on the single line function would be hit during the execution of any statement within the function; likewise, the breakpoint of the original source will generate breakpoints for all lines the function expands to, achieving the same functionality.

The main source breakpoints reside within the IDE, while the association of line mappings for the compilation stage sources relies on AST data and can thus take place only within the compiler. Naturally, some additional interaction is required to combine this information in order to generate the stage breakpoints.

Our initial approach towards this involved supplying the compiler with the original breakpoints, e.g. as arguments in its invocation. Then, during stage source generation, the computed line associations would be applied on them to generate the stage breakpoints. Finally, the resulting breakpoints would be sent back to the IDE, e.g. by

utilizing the communication channel between the backend and the frontend. The idea behind this approach was to minimize the data communicated to the IDE by keeping the line associations as internal compiler data available only during debugged compiler invocations. Nevertheless, line associations may also be required externally by IDE components, while they may be needed even after the compilation has been completed. For example they may be needed to provide better navigation across stage sources and main program transformations or to generate breakpoints for the runtime debugging of the final source deploying a similar method with the one used for stage breakpoints. To support this functionality, we adopt the original approach as follows. The line associations for compilation stage sources are normally calculated within the compiler during the unparsing and are then propagated to the IDE as accompanying metadata. Upon receiving a compilation stage source, and only in case of a debugged compile session, the IDE is then responsible to apply the line associations to generate the stage breakpoint based on any breakpoints present in main source. Finally, as in the original approach, any breakpoints generated this way are essentially transient, so they are only kept during the execution of their respective stage and discarded afterwards.

Apart from allowing the IDE to be aware of the line associations, the new approach also provides better modularity and separation of concerns. The compiler's only responsibility is now the generation and provision of the compilation stage sources and their corresponding line associations. In the IDE, the line associations now become part of the compilation stage source metadata and are maintained by the workspace management. Additionally, the generation and bookkeeping of the stage breakpoints is now part of the typical breakpoint management of the IDE. Finally, the debugger communication now requires no extensions to support posting a breakpoint from the backend directly to the frontend.

5.4.3 Enabling AST Inspection

The execution of a compilation stage typically targets the modification of the main source being compiled by manipulating code segments in AST form. To debug such operations, it should be possible to inspect such runtime values and browse through their contents. For example, using a typical expression tree-view the programmer should be able to navigate across a tree hierarchy representing an AST node and

inspect any of its attributes (e.g. type, name, value, etc.) or its related tree nodes (e.g. children, parent, etc.). Supporting such a facility may be directly supported by the existing debugger backend infrastructure, for instance if the AST nodes are represented using native data structures, or it may require some minor extensions.

While the above view may provide all relevant information for an AST, it becomes more and more difficult to use as the tree size grows. This is because the AST is visualized only through its root node and specific nodes within the hierarchy can only be viewed after navigating across all other intermediate nodes. For a better visual representation that directly illustrates the entire AST we propose using graphical tree visualizers. This way, programmers may simultaneously observe all AST nodes, the connectivity and relations among them as well as their specific attributes (directly annotated on the visualized tree or available as tooltip information). Since ASTs represent source code, another viable solution is to unparsed the AST into source code and provide it to the programmer with proper formatting and syntax highlighting, for instance reusing the editor component to deliver the view as a read-only document. In the latter case, additional handling may be required to properly visualize all possible AST values as they may also contain incomplete code segments. Overall, the IDE should ideally offer multiple alternative visualization schemes that programmers may select based on the nature of the program being debugged. For example, Figure 5.10 provides the alternative visualizations for inspecting the AST `<<x = 10>>`.

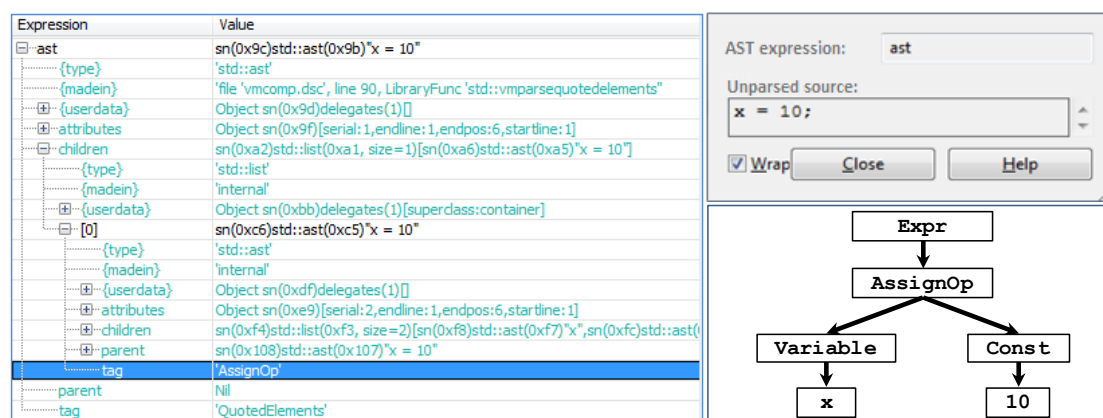


Figure 5.10 – Alternative views for inspecting AST values in Sparrow: an expression tree view (left), a viewer unparsing AST to source code (left, top) and a graphical tree view (left, bottom).

5.4.4 Study for Runtime Metaprogramming

Again, the above discussion was focused towards compile-time metaprogramming. We continue by exploring how stage debugging can be achieved in a language offering runtime metaprogramming and discuss directly applicable existing practices, their limitations as well as possible extensions. As with the compile-time case, we focus on the requirements for stage debugging (i.e. source information and debug information) as well as the support for setting breakpoints in the context of the stage metaprogram. We begin with languages that support metaprogramming through reflection and again use *C#* and *Java* as examples.

5.4.4.1 Stage Debugging

As previously discussed, in both *C#* and *Java*, the compiler can be used at runtime to compile any dynamically created source text. The compilation outcome is typically a binary file (dll or exe file in *C#*, class file in *Java*) that can be loaded and executed using the reflection API. Deploying such code within a typical debug session and ensuring that the proper options for generating debug information are supplied to the compiler invocation, we end up with a normally executing binary that can be debugged. However, we are missing the source information required for true source-level debugging. In *C#*, compilation is performed through a temporary file containing the dynamic source text. Using a compiler option it is possible to specify that the temporary file should be retained after the end of the compilation. This way it can be automatically used by the debugger to trace execution of the dynamically generated code.

In *Java* there is no similar option to automatically generate a source file to be used for debugging purposes. Nevertheless, an existing source file matching the generated binary file can be automatically loaded and used for source-level debugging. This means that it is possible to achieve the desired functionality by manually storing the dynamic source text in an appropriate source file before loading the generated binary file. To automatically support this functionality without requiring any intervention from the programmer, we propose utilizing the information present in the class file to generate a source file through reverse engineering, i.e. using a *Java* decompiler (e.g. [Dupuy]). To this end, the *Java* Platform Debugger Architecture [Oracle] would have to be extended with an extra command allowing the debugger frontend to ask the

backend for missing source information, while their interplay would be as follows. The backend initially issues a stop-point at a specific source location. The frontend, typically part of the IDE, checks if the target source is present in the project management and if not, it asks the backend to supply a source file. The backend that has access to the generated class file, invokes the decompiler to generate a matching source and sends it back to the frontend. Finally, the frontend receives the source and opens it in an editor to support source-level debugging. From that point, the source file can be used as any normal file, allowing the programmer to add extra breakpoints and supporting typical debugging facilities.

Apart from using the compiler on a dynamic source text, it is also possible to directly emit code in intermediate or final form. In C# there is the `Reflection.Emit` namespace containing functionality for generating intermediate language code. When emitting code this way, it is possible to associate it with a source file to be used for debugging. However, this involves manually specifying a corresponding source location for each emitted instruction as well as explicitly providing names for any generated symbols. Standard Java libraries do not provide a similar functionality; however it is possible to generate byte-code using a third party library like the *Byte Code Engineering Library* [BCEL] or *ASM* [Bruneton]. The generated binaries can be loaded for execution during a typical debug session, but they do not provide an associated source representation. Nevertheless, with the class file present, the proposed method of decompiling the class applies here as well, allowing the creation of a source file that can be used for debugging purposes.

A reflection infrastructure is also offered by the Delta language. To provide source information for debugging purposes when the original code is stored in a buffer or when only a respective syntax tree is available for translation we use the following approach: The source text is incorporated into the debug information of the generated binary. Once the binary is loaded for execution, the source text from the debug information is extracted by the debugger backend and is posted to the debugger frontend when a breakpoint is hit in a statement of the dynamic source code. Then, the frontend opens an editor for the dynamic source code enabling users to review it and also add or remove breakpoints as needed.

In general, the data required to carry out stage debugging consist of the stage source text and the respective debug information. Typically, they are utilized by different components during the debug process and target different tasks; the stage source is used for displaying code and tracing through it as well as setting breakpoints while the debug information is used for expression evaluation and call stack information (Figure 5.11, right part). As such, they may be stored independently, both either in memory or as a files within the filesystem, effectively presenting different options regarding their availability. Supporting each of these options is important towards allowing source-level debugging for all cases; however existing languages typically provide only partial support. For example, in C#, it is possible to compile a source present in memory as well as a source present in the filesystem but the generated binary and debug information file (pdb file) is always stored in the filesystem. Even if we explicitly set that generation should take place in memory (using the `CompilerParameters.GenerateInMemory` property), temporary files are always created in the filesystem and are disposed after compilation. This intermediate step may in fact cause unexpected errors to occur if for example the disk quota is exceeded or the application lacks access privileges. Even worse, during a subsequent debug session, the debugger will request these intermediate files to provide source and debug information, however they have been already disposed after compilation (unless explicitly otherwise specified), meaning that no proper debugging is possible. Similarly, in Java, the debug information is inserted into the generated class file that is always stored in the filesystem and the debugger always requires a source file in the filesystem to match that generated class file. Essentially, in both cases, the only possibility for a proper debug session is when all required information is available in the filesystem. In Delta, where the dynamic source is part of the debug information and the debug information itself is stored within the generated binary it is possible to have that binary stored in both filesystem and memory. In particular, regardless of the stage source text or generated binary presence (either memory or filesystem), Delta provides full source-level debugging support.

The left part of Figure 5.11 summarizes the different availability options for stage source text and its respective debug information and the debugging support offered by each language discussed.

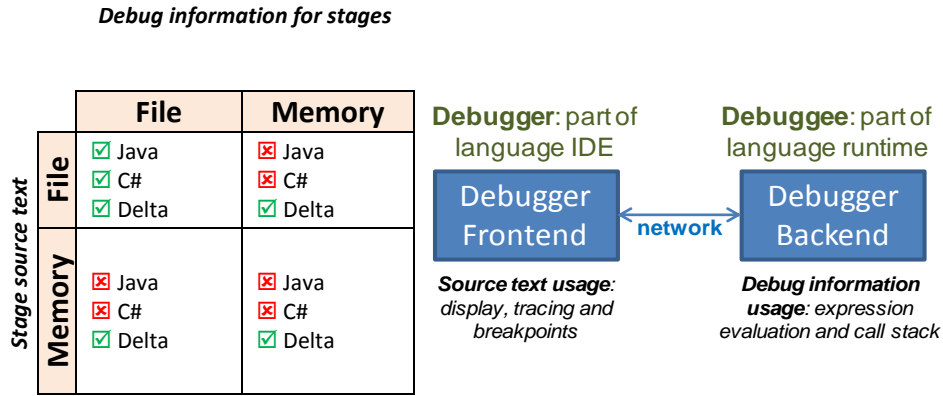


Figure 5.11 – *Left:* source-level debugging support for runtime stages in C#, Java and Delta regarding the different availability options of stage source text and its respective debug information; *Right:* the split responsibility in using source text and debug information for debugging sessions.

5.4.4.2 Stage Breakpoints

Once a source file corresponding to the metaprogram being executed becomes available, it is possible to use it for setting the desired breakpoints. However, the previously discussed methods for providing the source file imply that execution is already stopped within the metaprogram context. Effectively, we need a way to set breakpoints in the generated code *before* it becomes available, or an entry point for breaking execution within the generated code and then placing additional breakpoint directly on its source.

The simplest scenario is to locate the first invocation targeting generated code and use that as an entry point. Within the debug session, we can place a breakpoint at the already present source that will invoke the generated code and then step into that invocation to cause execution to break in the desired context. However, this may not always be possible as the generated code is not invoked necessarily directly after loading. For instance, the generated code may contain callbacks that are only registered upon loading and whose invocation occurs at an unspecified time later in the execution of the program. In such cases one possible option is to instrument the generated code with explicit directives for breaking system execution (granted of course that the language and execution system provide such a facility). For example, such functionality is provided in C# through `System.Diagnostics.Debugger.Break`. In case of a debugged execution (that can be determined at runtime through `System.Diagnostics.Debugger.IsAttached`) it is

possible to insert such break invocations where needed in the code to be generated. For the callback example mentioned earlier, that would involve placing a break invocation as the first instruction in each of the callbacks present in the generated code. In general, the same would have to be done for every executable piece of code (e.g. function, method, etc.) present in the generated binary. The latter may result in multiple breakpoints being issued with subsequent invocations of generated code, while the original intent was only to break execution in the first invocation. However, as shown in the code example of Figure 5.12 this is easily resolved by substituting the original call with a wrapper that only breaks execution in its first invocation.

<pre> string breakDef = ""; string breakCall = ""; if (System.Diagnostics.Debugger.IsAttached) { breakDef = "static bool firstTime = true;" + "static void BreakWrapper() {" + " if (firstTime) { firstTime = false;" + " System.Diagnostics.Debugger.Break(); } }"; breakCall = "BreakWrapper()"; } StringBuilder sb = new StringBuilder(); sb.AppendLine("class Test {"); if (breakDef.Length > 0) sb.AppendLine(breakDef); sb.AppendLine("public void func1() {"); if (breakCall.Length > 0) sb.AppendLine(breakCall); sb.AppendLine(" System.Console.WriteLine(\"func1\");"); sb.AppendLine("}"); sb.AppendLine("public void func2() {"); if (breakCall.Length > 0) sb.AppendLine(breakCall); sb.AppendLine(" System.Console.WriteLine(\"func2\");"); sb.AppendLine("}"); string source = sb.ToString(); // use 'source' for compilation hereafter... </pre>	Original file
<pre> class Test { static bool firstTime = true; static void BreakWrapper() { if (firstTime) { firstTime = false; System.Diagnostics.Debugger.Break(); } } public void func1() { BreakWrapper(); System.Console.WriteLine("func1"); } public void func2() { BreakWrapper(); System.Console.WriteLine("func2"); } } </pre>	Generated file for debug session
<pre> class Test { public void func1() { System.Console.WriteLine("func1"); } public void func2() { System.Console.WriteLine("func2"); } } </pre>	Generated file for run session

Figure 5.12 – Example in C# illustrating the instrumentation of the generated code with debugger break instructions to stop execution in the context of the generated code during a debug session.

While the above is an adequate solution to the problem, it requires introducing additional code just for debugging purposes, something typically undesirable. The original problem narrows down to breaking execution after loading the binary and before executing any of its code, so it should be addressed as a breakpoint issue. Apart from normal breakpoints, there are also conditional breakpoints, breaking execution when a condition is met, data breakpoints, breaking execution when the value of some data is changed, and exception breakpoints, breaking execution when an exception is thrown. In the same sense, we propose introducing a breakpoint category dedicated for breaking execution upon loading a binary file. In the presence of such a breakpoint, when a generated binary is loaded, execution will be stopped and the debugger will generate a matching source file using one of the earlier

discussed methods. This source file can then be utilized to add any normal breakpoints directly in the generated code before it is executed, thus solving the initial problem without requiring code instrumentation and with minimal debugging effort from the programmer (essentially only enabling the proposed breakpoint when necessary).

5.4.4.3 Multi-Stage Languages

Multi-stage languages that support runtime code generation through staging annotations (e.g. *MetaOCaml*, *MetaML*, *Metaphor* [Neverov04][Neverov06], *Mint*) could also use a similar approach to the one discussed for compile-time metaprogramming. In such languages, the stage code is typically available in some AST form that can be unparsed to provide a source file to be used for debugging. Regarding breakpoint support, it should also be possible to associate lines from the original source to lines of the generated source to automatically generate breakpoints for the stage program. Nevertheless, there is a significant difference compared to the compile-time case. Since breakpoints are typically targeted for runtime execution, they do not interfere with the compilation of the original program; however, if code generation occurs at runtime the same breakpoints may be triggered by the normal execution flow of the program even if the intent was to use them just for generating breakpoints for the stage program. The latter essentially demonstrates the need for disambiguating between breakpoints targeted for normal program execution and breakpoints targeted for stage metaprograms. In this sense, we propose introducing a new breakpoint category for *explicit stage breakpoints*. Such breakpoints may be set in the original source just like normal breakpoints but they do not cause execution to break; instead they are only used in the context of staging and allow generating normal breakpoint for all stage programs based on the previously discussed line associations.

In particular, multi-stage languages based on top of a language that supports reflection, like *Mint* (based on *Java*) or *Metaphor* (based on *C#*) may utilize the reflection mechanism and the approach mentioned previously to support stage debugging. Such a language could be implemented using a stage preprocessor, an AST composition library and the pure base language along with its support for dynamic compilation (Figure 5.13). Initially, the source file that includes staging

annotations is provided to the stage preprocessor that translates them into the appropriate AST composition invocations, while associating them with information about the source and line they originate from. If the run construct is syntactic (e.g. `run code` as in MetaML) it is preprocessed as well, translating them into invocation of the reflection API that will dynamically generate code from the constructed AST (either directly or by first obtaining its source text through unparsing). Otherwise, if the run construct is a normal language invocation part of the AST library (e.g. `code.run()` as in Mint), then the invocation of the reflection API is already present in the implementation of the run method of the AST library. In both cases, the reflection API will be used at runtime to dynamically generate and execute the stage code, thus allowing it to be debugged with the existing language infrastructure and extensions we discussed in the previous sections.

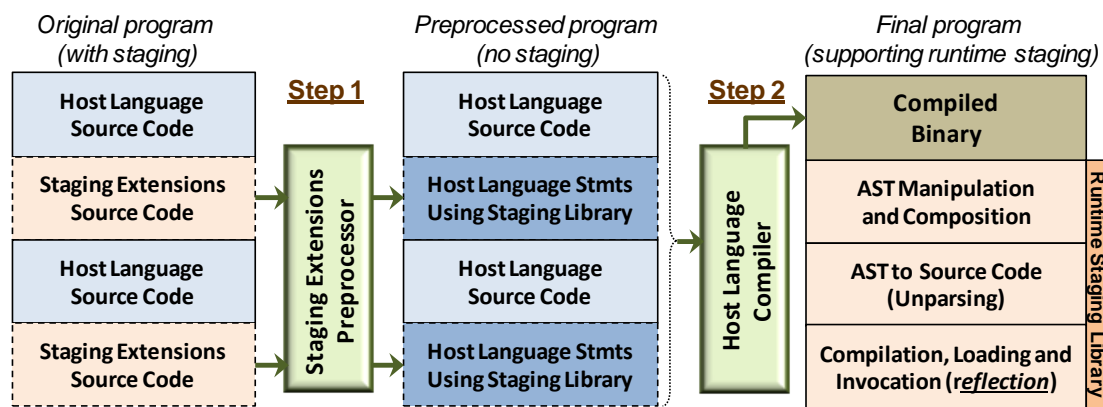


Figure 5.13 – Sample implementation of a multi-stage language with runtime metaprogramming relying on: (i) staging extensions preprocessing; and (ii) runtime code generation, loading and invocation via the language reflection facilities.

Chapter 6

Support for Aspects

“Nevertheless, I consider OOP as an aspect of programming in the large; that is, as an aspect that logically follows programming in the small and requires sound knowledge of procedural programming.”

-Niklaus Wirth

Aspect-Oriented Programming (AOP) [Kiczales97] is a methodology for modeling crosscutting concerns into modular units called *aspects*. Aspects contain information about the additional behavior, called *advice*, that will be added to the base program by the aspect as well as the program locations, called *join points*, where this additional behavior is to be inserted based on some matching criteria, called *pointcuts*. Aspects are typically expressed in separate languages and an *aspect weaver* combines the base program with the aspect program to form the final program.

In the context of multi-stage languages, AOP could be used not only for normal programs but also for *all stages* they might contain. As thoroughly discussed in previous chapters, stage metaprograms, besides their special mission being primarily generative to produce code, are essentially no different to normal programs. Thus, they deserve, and require, *all typical programming techniques of normal programs, including aspects*. For instance, within stage code, one may deploy logging aspects to support tracing of method invocations, or apply exception handling aspects at appropriate call sites. Apparently, there is no particular reason to forbid the application of such aspects in stage code. In fact, there are also various scenarios related to the generative role of stages. For example, stages typically handle code in the form ASTs, so we could define aspects for AST manipulation, such as decorating with extra code, validating according to criteria, or introducing custom iteration policies.

Without aspect support we simply limit the potential for developing stages using state of the art programming practices. For instance, consider *AspectJ* [Kiczales01], a

popular language for AOP, and *Mint* [Westbrook], a Java extension offering staging facilities. Staged code within a *Mint* program is actually Java code. However, it is not possible to use AspectJ to apply AOP on the staged code as it is never available in a form that can be manipulated by the aspect weaver. In fact, the reason is more fundamental: *no interplay between the aspect weaver and a staging system has ever been considered or proposed.*

In current implementations for AOP the *language compiler* is ignorant of the *aspect weaver* and the transformation it performs on the functionality of the original program. Overall, the aspect weaver is *never* in the compilation or execution loop. However, in multi-stage languages, stages are composed and evaluated during either compilation or execution disabling any possibility for the aspect weaver to intervene. Additionally, the source or binary of a stage is transient during compilation or execution and cannot be available to the aspect weaver unless it becomes part of the loop. To resolve this, the respective source or binary files for stages must be created and supplied to the aspect weaver during the staging process. For the previous AspectJ and *Mint* example, the latter would involve explicitly writing aspects for staged code and weaving them into the binary along with the staged code so that the stage evaluation contains the advised functionality.

In this direction, we propose the adoption of AOP in multi-stage languages, introduce a methodology for aspect weaving in the entire staging pipeline and discuss an implementation on the multi-stage extension of the Delta languages. In particular, we do not introduce a separate aspect language for AOP, but we implement aspects as batches of AST transformation programs written in the same language. This approach fits well with typical multi-stage metaprogramming practices since programmers are already familiar with using and manipulating ASTs. Also, it allows exploiting features like reviewing, inspecting or debugging AST transformations that may already be offered by the multi-stage language IDE. Despite the particular implementation, the two distinct methodologies, i.e. supporting aspects for stages and applying them without dedicated languages are orthogonal and can be deployed independently of each other. In fact, we also discuss how aspects for stages can be supported in a mainstream AOP language like AspectJ.

6.1 Aspects for Stages

There are two approaches for weaving aspect code along with normal program code: *source-level weaving* and *binary-level weaving*. Source-level weaving involves applying the aspect on the original source to get the transformed version of the source that is then compiled to binary (Figure 6.1, top). On the other hand, in binary-level weaving the source is normally compiled to binary and then the aspects are applied to generate an updated binary version (Figure 6.1, bottom). However, in the context of existing multi-stage language implementations, none of these approaches is sufficiently supported to facilitate AOP at a full scale. To explain why, we first consider the potential options for applying aspect weaving (either source- or binary-level) being *before*, *during* and *after* the staging process. Then we study the way such options can be supported under both CTMP and RTMP, the latter either for compiled implementations - $RTMP_C$, or interpreted ones - $RTMP_I$ (clearly, the distinction refers to the implementation method, not the language itself). As we discuss latter, the weaving options, and the way they can be applied, strongly depend on the implementation approach of the target multi-stage language. Although our system supports CTMP and source-level weaving, we discuss all possible combinations to outline the differences involved towards weaving implementation, and also cover a wider range of languages. For example, RTMP concerns mainstream languages like *Java* and *C#*. Although not multi-stage languages by default, they have extensions that introduce multi-stage programming constructs, and also provide powerful reflection mechanisms that enable some degree of runtime metaprogramming. Overall, the multi-stage language implementation approach maps to different options of source or binary weaving, which display varying usability, expressiveness and efficiency properties when it comes to programming and applying aspects.

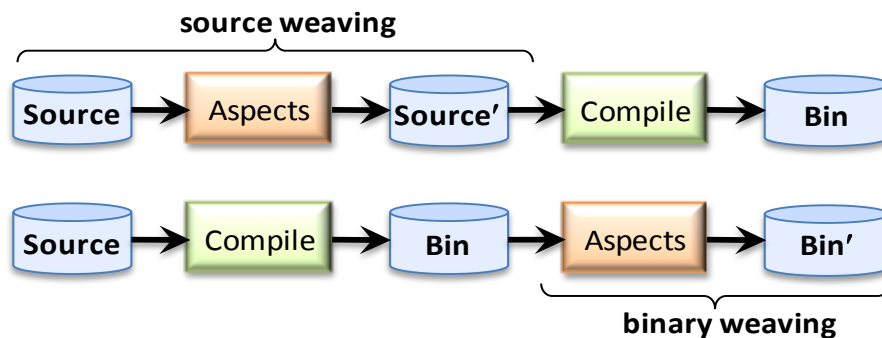


Figure 6.1 – The two alternative contexts for aspect weaving.

6.1.1 Weaving Options

We revisit the processing diagrams of the various staging approaches highlighting the potential options for applying aspect weaving in the context of multi-stage languages. These options are not mutually exclusive and can be combined to achieve aspect orientation in multiple steps of the compilation or execution process.

In CTMP, we can apply source-weaving on the initial source (Figure 6.2: 1) before it is parsed into the AST form that will be used for the staging process. Then during staging, we can apply either source- or binary-weaving respectively on the source or binary form of each stage metaprogram (Figure 6.2: 2-3). Finally, when no more stages exist, we can again apply source- or binary-weaving on the final version of the code, as transformed by all stage evaluations (Figure 6.2: 4-5). Of course, to apply source-weaving either during or after the staging process, we first need to unparse the respective AST into a source file; the transformed source resulting from the weaving process can then be compiled to produce the respective binary code (either stage binary or final binary).

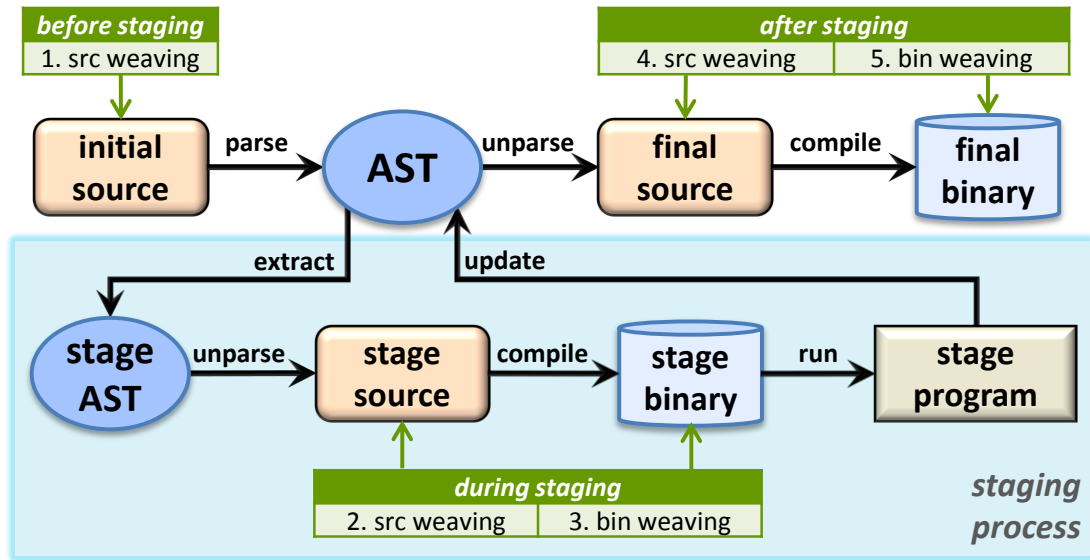


Figure 6.2 – Compile-time staging and aspect weaving options.

In RTMP_C, we can apply source-weaving directly on the main source and then compile it to binary form or perform the compilation first and then advise it through binary-weaving (Figure 6.3: 1-2). The same options apply also for the dynamic source and its binary that are generated as part of the staging process at runtime. This way we can apply either source- or binary-weaving for all stages involved (Figure 6.3: 3-4).

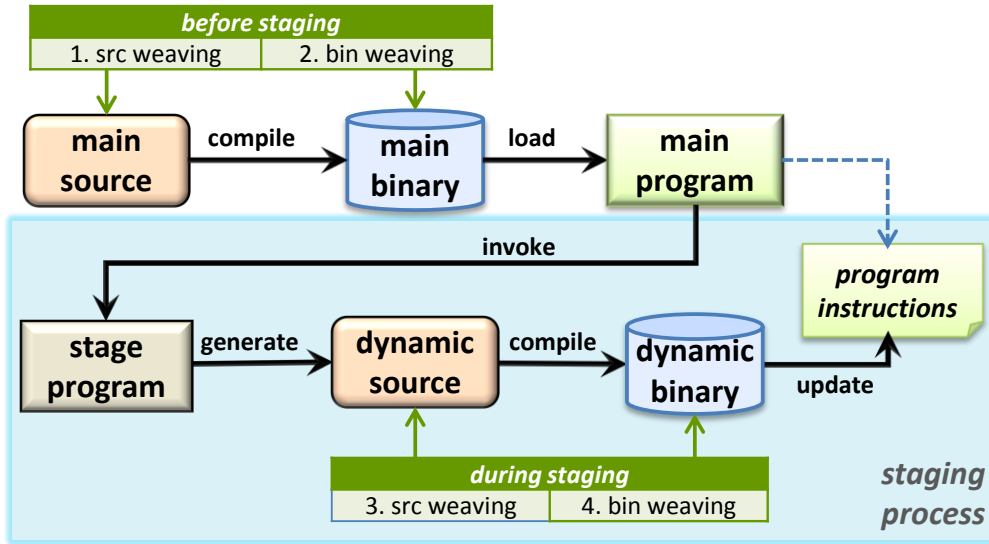


Figure 6.3 – Runtime staging (compiled language case) and aspect weaving options.

In $RTMP_I$ we can apply source-level weaving on the initial source (Figure 6.4: 1) before it is parsed to AST form and sent to the interpreter for evaluation. Then, after extracting stage code and before evaluating it, we can unparse it to get the respective stage source, apply source-weaving on it (Figure 6.4: 2) and then reparse it to get the transformed AST that will be evaluated recursively. for any stage code the stage source that has to be unparsed for this purpose.

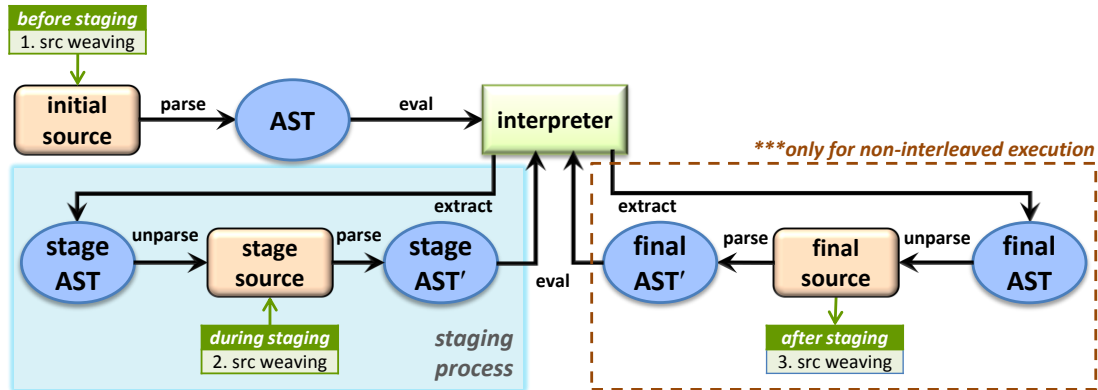


Figure 6.4 – Runtime staging (interpreted language case) and aspect weaving options.

To offer the weaving option *after* the evaluation of stages (Figure 6.4: 3) a small modification on the way stages are actually interpreted is required. In particular, stages are commonly evaluated as part of the main program execution and as soon as they are met within program definitions. Thus, staging evaluation interleaves main program evaluation. In general, once staging completes, part of the main program is already executed, rendering meaningless to apply aspects on a program that is already

partially evaluated. The reason is simple to explain. Consider we allow such weaving to take place, and imagine a function that is affected by weaving and which has already been invoked many times. Then, the semantics of such a function can vary during execution, with the version before weaving being different to the one another after weaving. Now, this sort of inconsistency appears only due to interleaving of stage evaluation with the main program. To adopt an interpreted evaluation that disables interleaving would be trivial in RTMP₁ implementations and with no discount on stage expressiveness. In particular, it suffices to apply stage evaluation first, and then, once no staging remains, proceed with the evaluation of the main program. In fact, this type of ordering is similar to CTMP implementations, while setting *after* staging weaving a well-defined option.

Notice that in all previous cases, the initial source file contains the normal program code together with stage code, while aspect code is considered to be in separate files. Thus, a weaving process may in fact apply aspects to any of them. We further elaborate on what functionality can be addressed by the potential aspect applications in the following section.

The difference between applying AOP in a normal language and a metalanguage is essentially that in the former case, there is a single source or binary for transformation, while in the latter case there are multiple sources or binaries for transformation and they are involved in different parts of the process. Table 6.1 summarizes the options for applying AOP with different combinations of source and binary aspect weaving for each of the staging approaches.

Table 6.1 – Ability to implement aspects under different categories of multi-stage languages and for the different possible weaving contexts and subjects

Weaving Context	Weaving Subject	Compile-time Staging	Runtime Staging	
			Compiled language	Interpreted language
Source Code	Initial source	✓	✓	✓
	Stage sources	✓	✓	✓
	Final source	✓	N/A	✓
Binary Code	Initial binary	N/A	✓	N/A
	Stage binaries	✓	✓	N/A
	Final binary	✓	N/A	N/A

Trying to apply the current AOP practices without interfering with the staging pipeline, means essentially operating as a source code pre-processor or binary code post-processor, thus limiting the potentials for aspect weaving. For CTMP we are limited to weaving options 1, 4 and 5, for RTMP_C we are limited to weaving options 1 and 2, while for RTMP_I we are limited to the single option 1. These however cannot fully express aspect transformations in the staging pipeline. For RTMP_C, this should be clear, as the dynamic code is generated at runtime with no way to be updated. For RTMP_I, it would be possible for the normal or staged code present in the initial source but there is no way to handle any code introduced by staging. For CTMP, the only supported scenario relates to a two-stage language, where we have only one stage of metaprogramming and the entire meta-code is available within the original source. In this case, it is possible to apply source-level weaving to transform the existing meta-code, while also applying binary-level weaving right after compilation to transform the code generated by the metaprogram. An example for this scenario would be C++, where a pre-compilation source-level weaving could transform the template code (i.e. the stage-code) and a post-compilation binary-level weaving transform the template instantiations (i.e. the generated code). The previous method cannot be applied for languages with more than two stages (i.e. more than one nested metaprograms). The reason is that the initial source-level weaving can only transform the meta-code that already exists in the original source, but not the meta-code that is introduced as a result of a previous stage. Other than that, any binary-level weaving would operate on the final program source after all stage metaprograms have been executed, and of course cannot transform their functionality.

Another possible weaving approach, still operating on binary level, would be to insert the extra functionality upon loading of the binary. This can be achieved by extending the loader with hooks that will perform the weaving, being the way load-time weaving is actually supported in AspectJ. Disregarding any performance penalties about the computations taking place at loading time, binary loading occurs for both normal and stage programs, so this approach could potentially be used to weave functionality in both of them without interfering with the staging pipeline. However, this method does not allow differentiating between normal and stage programs, meaning they cannot be supported with different aspects. The only way enabling different aspects, following our proposition, is for the multi-stage language to uniquely name and separate the

produced stage classes for all stage code fragments. The later would allow load-time weaving approaches by selectively intervening only on stage classes, once adopting the class name patterns of the language compiler. However, the latter requires two important changes. Firstly, we should guarantee that the multi-stage language generates separate classes for stage code snippets, something not currently supported by known runtime multi-stage languages for Java. Secondly, the naming patterns for stage classes should become a documented language feature so that load-time weavers can exploit them. Essentially, these two extensions serve no other purpose than allow bringing a load-time weaver into the staging loop. The later repeats our earlier argument that *no stage-level weaving is possible without the multi-stage language actually setting the ground*. Additionally, load-time weaving is applicable only for languages compiled to byte-code, like Java or C#, when run directly by respective virtual machines. However, it is not appropriate when Ahead-Of-Time compilation (AOT) is applied on such languages. Clearly, it is not applicable for languages that directly generate native code, like C or C++. Overall, we consider load-time weaving to be insufficient for full-scale aspect deployment within a multi-stage language and do not further include it in our discussion, although it could be used to achieve similar functionality with some of the case studies discussed later.

In conclusion, in order to effectively support aspects for stages in a multi-stage language, aspect weaving should be necessarily introduced as part of the staging process.

6.1.2 Aspect Categories

In a multi-stage language, the original program p_0 also contains the various stage metaprograms s_1, \dots, s_n . With the execution of these stages, the original program p_0 is transformed sequentially to p_1, \dots, p_n , the last being the final program version. In AOP, we typically have the original program p that is advised by the aspect program a . Introducing AOP in a multi-stage language requires considering the various interaction points: (i) program p_0 is advised by aspect program a ; (ii) stage metaprograms s_1, \dots, s_n are advised by aspect program a ; and (iii) intermediate program transformations p_1, \dots, p_n are advised by aspect program a .

Considering the first interaction point, the program p_0 contains both normal program code and staged code, meaning that the aspect a could advise any of them. However,

none of them have their final form yet; normal code may be transformed by stage code, while code of a particular stage may be transformed by higher stage code. This means that applying aspect a to advise normal program code or stage code may cause inconsistencies and thus should be avoided. For example, consider a scenario where we advise the normal code to insert logging functionality for all functions it contains. With this taking place before the staging process, any functions generated due to staging will not contain the logging functionality, eventually resulting into final code where only some of the functions are actually advised. Nevertheless, applying an aspect on the original program can be useful. It can introduce additional code for a specific stage or even introduce extra stages. Such an aspect can be seen as a higher-order metaprogramming facility that allows the transformation logic to be entirely decoupled from the main source code. For example, this allows turning normal code to stage code to perform some computations during compilation and improve performance (sort of partial evaluation) or introduce stage code that performs static analysis on specific parts of the original source. Such aspects are always executed before the staging process, so we call them *pre-staging aspects*.

In the second interaction point, we have each stage metaprogram s_i being advised by the aspect program a . Each stage contains code from both the original program along with code generated by stages directly embedded in it (higher-order). Thus, applying the aspect right before its evaluation guaranties that the stage has its final form and that the advice functionality is consistent. The reason for using such aspects relates to crosscutting functionality typically found in stage code. With stages involving code generation, the manipulation of ASTs is very common, typically involving scenarios of structural validation, decoration with extra functionality or attributes, and custom iterators. Apart from their special purpose as code generators, stages are also programs that may involve crosscutting functionality typically found in normal programs, like synchronization, logging and monitoring. For instance, a common scenario may involve adding logging calls to trace meta-function invocations. For the weaver to deploy such aspects, it needs to have access to the source or binary code of each stage. This means that the compiler (in CTMP) or the runtime system (in RTMP) should not treat stages as private transient programs, but should somehow supply produced source or binary files to the aspect weaver to operate on. Additionally, interplay between the weaver and the compiler or runtime is required

following the actual weaving process. More specifically, in source-level weaving, the compiler (or interpreter) generates the stage source, gives it to the weaver and gets back the advised version that it then compiles (or interprets). In binary-level weaving (only in compiled-languages), the compiler first compiles the stage source to binary, gives it to the weaver and gets back the advised binary version. We call such aspects *in-staging aspects*.

Regarding the third interaction point, we notice that the intermediate program transformations p_1, \dots, p_{n-1} are in fact intermediate forms. This means that any aspect application in them occurs on an incomplete program and may thus cause inconsistencies. The case of applying an aspect on p_n in particular requires that all stage evaluations have been performed and that there is no more staging involved in the final program version.

It should be noted that in RTMP, either interpreted or compiled, it is generally undecidable to judge if no further staging process can take place after a certain runtime point. The reason is that use of reflection mechanisms, dynamic loading or *eval* can generate implicit staged code, not visible in the currently executing program instructions. Moreover, in either RTMP or CMTP, aspects applied *after* staging could also introduce further staging. Consequently, there is no way to impose just a single staging process. As a result, we define as *final* a program containing no more staged code. Clearly, if implicit staging is introduced by the evaluation of the *final* program itself, or by aspects applied *after* staging, then further aspect weaving following the proposed approach reapplies. With such a scenario, additional staging rounds occur, leading to another *final* program at the end. In this sense, the term *final* just denotes the program resulting from a staging process, not by all staging processes. Overall, aspects on p_n play the same role as aspects on normal programs: there is a program that needs to be advised involving no staging. They are applicable to both CMTP and RTMP_I, for the latter assuming a non-interleaved execution. They do not apply to RTMP_C, since, when the runtime staging process completes, part of the program has already been executed. Such aspects are always applied after the staging process, so we call them *post-staging aspects*.

Table 6.2 gives an overview of each discussed aspect category, highlighting its purpose and deployment options for each staging approach.

Table 6.2 – Ability to implement aspects under different categories of multi-stage languages and for the different possible weaving contexts and subjects

	Purpose	Compile-time staging	Runtime staging	
			Compiled language	Interpreted language
Pre Staging	Introduce or update staging on the original program	Before source compilation (<i>source weaving</i>)	Before main compilation (<i>source weaving</i>) or after main compilation (<i>binary weaving</i>)	Before main interpretation (<i>source weaving</i>)
In Staging	Update stages	Before stage compilation (<i>source weaving</i>) or after stage compilation (<i>binary weaving</i>)	Before dynamic source compilation (<i>source weaving</i>) or after dynamic source compilation (<i>binary weaving</i>)	Before stage interpretation (<i>source weaving</i>)
Post Staging	Update the final program	Before final compilation (<i>source weaving</i>) or just after compilation completes (<i>binary weaving</i>)	N/A (main program is already executing)	After non-interleaved interpretation of all stages (<i>source weaving</i>)

For a complete combination of stages and aspects one may introduce multi-stage programming in the context of an aspect program. In fact, multi-stage languages fully support nested stages, being metaprograms that generate the code of enclosing metaprograms. Similarly, one could consider chained aspects, being aspects applying cross-cutting concerns on the logic of other aspects. Thus, their combination is theoretically unlimited. Regarding the blending of stages and aspects, an aspect program a_0 may itself contain stage metaprograms s_1, \dots, s_n that transform it sequentially into a_1, \dots, a_n , the last being the final version of the aspect program. Such a combination is meaningful, enabling aspect properties like pointcuts and advice to be generated through metaprogramming. However, it requires the aspect language to be extended with extra constructs (e.g. as in [Zook]). If we consider an aspect to be applied to a client program through a binary executable form, any staging during aspect compilation is transparent to all of its clients, so its deployment remains the same despite staging. Effectively, the two sides of the combination between multi-stage languages and AOP are orthogonal and can be adopted independently of each other. In this thesis we primarily focus in the first direction, i.e. introducing AOP in a multi-stage language. However, as discussed in the following section, we provide

aspects as transformation programs written directly in our multi-stage language, meaning our aspects can be fully staged.

6.2 Aspects without Dedicated Languages

A spin-off outcome of our work is an alternative way to apply aspect transformations. Essentially, we treat aspects as AST transformation programs written in the same language and deploying an aspect library working on ASTs. We continue by elaborating on this notion and discuss how such transformation programs can be integrated in the workspace management and build process of the integrated development environment. We do not argue that this is the ultimate approach towards supporting AOP; we present it as a viable alternative and discuss its advantages when deployed in an existing multi-stage language.

6.2.1 Aspects as Transformation Batches

To test aspects for stages we started thinking of crafting a prototype aspect engine for our staged language. In this context, we observed that the language offers quasi-quotes, escaping, and a comprehensive AST library, all of which are not staged but can be used as part of a normal program. Now, the latter are essentially everything one needs to algorithmically perform source code transformations. Practically, aspects are a restricted form of algorithmic cross-cutting transformations, currently offered with distinct languages with automations in expressing pointcuts and advice. Regarding pointcuts, one might directly offer a library set to search AST nodes against criteria defined as predicate functions, or even through some custom string-based pattern matching language. A similar library set can also be offered for defining and applying advice.

The prototype implementation of our approach offers only static aspect transformations, being analogous to the static model offered by AspectJ. However, treating aspects as transformation batches is not limited to static weaving as such. Batches may be implemented in a runtime preprocessing process to perform binary or load-time weaving, thus operating in a way similar to the dynamic aspect weaving model. We focused on a compile-time static model only for practical reasons: (i) it is more efficient, as it introduces no runtime overhead; and (ii) it leads to smaller executable images, since the aspect program is not linked with the affected program.

Along these lines, it became clear that all aspect features may be directly realized via a respective aspect library working on ASTs. This led us to the idea of turning aspect programs to normal language programs taking as input the AST of another program while deploying the aspect library to apply pointcuts and advice directly in the main language. In particular, aspects programs contain a main function, conventionally called *transform*, which takes a single AST argument, transforms it as needed and returns the updated version. To apply a series of aspect programs on a source file we use a special weaver program. The weaver initially takes the source file and parses it into an AST. Then, for each of the given aspect programs, it invokes the *transform* function passing as argument the current AST version which it then updates based on the function's return value. The same process continues until all aspect programs have been applied and the source has been transformed to its final form, encompassing the source code as advised by all the aspects. Essentially, aspect weaving is a batch process of AST transformations (see Figure 6.5).

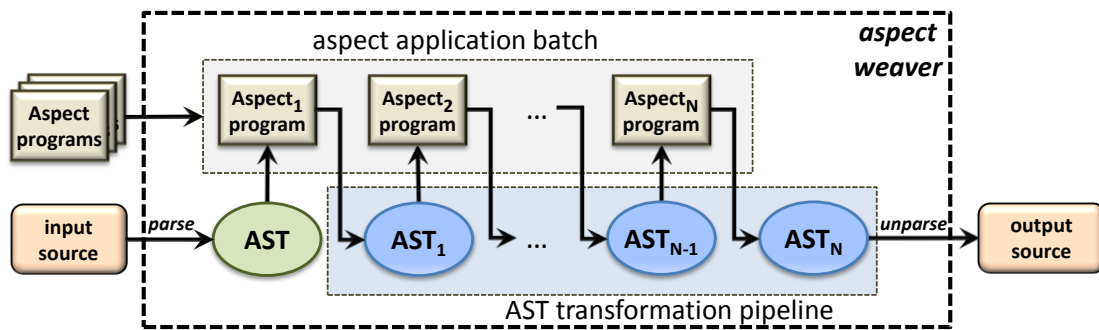


Figure 6.5 – Source-level aspect weaving as a batch of AST transformation programs.

The above transformation process applies to all discussed aspect categories. As shown in Figure 6.6, we essentially have three batches (chains) of transformation programs as follows: (i) one for the original program - *pre-staging aspects*; (ii) one for each stage - *in-staging aspects*; and (iii) one after the staging process - *post-staging aspects*. Once the last batch is applied, the final program version is then compiled to binary form.

From a deployment perspective, we try to minimize the coupling between the compiler and the aspect weaver and achieve a uniform invocation style. In fact, they never communicate explicitly, but are coordinated by the build system. To this end, the aspect weaver always receives a batch and an affected source file as input, and

produces a source file as output. In this sense, the weaver is unaware of the previously mentioned aspect categories. It simply applies the current transformation batch to the input source file. Along these lines, the compiler also receives a source file as input by the build system. With aspects present, the transformed source version is supplied to the compiler; otherwise the original source file is directly supplied. The reason for unparsing the result of every aspect transformation batch, and not maintaining it in the form of an AST, is for simplicity, since this way, we retain the original compiler accepting directly source text.

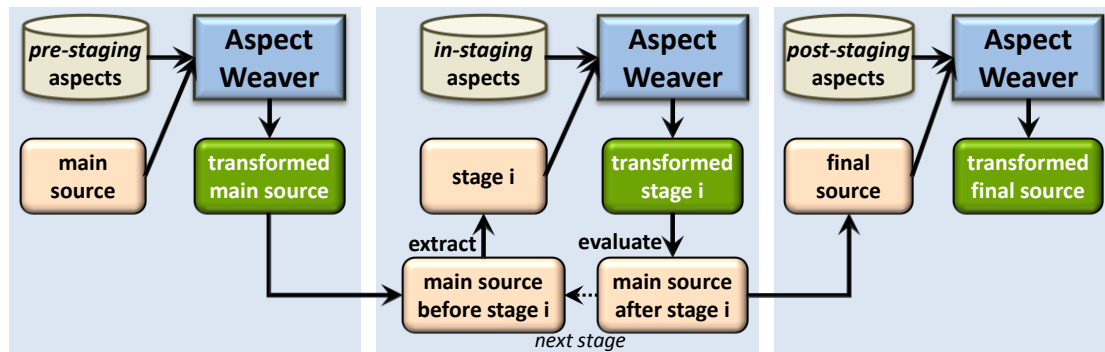


Figure 6.6 – Complete overview of all aspect transformation batches occurring during compilation.

This approach has many advantages compared to custom aspect languages. Firstly, no second language and translator are deployed. Secondly, by turning aspects to normal programs they can be directly hosted in the language IDE and be normally debugged. Thirdly, aspects become first-class IDE citizens and thus can be managed under the same umbrella with the programs they actually transform. Finally, their software engineering directly reuses the techniques and constructs of normal programs, not requiring reinventing the wheel as with aspects languages (e.g. aspect inheritance is essentially reintroduced). In the context of a metalanguage, the latter also applies for its metaprogramming features, effectively enabling metaprogramming support for aspect programs. For a homogeneous metalanguage in particular, having the metalanguage operate also as an aspect language offers great compositional flexibility and expressiveness: programs may be advised by aspects, these aspects may be further advised by other aspects or be subject to staging, any such stages may also be advised by aspects and so on.

Overall, we realized that the AST manipulation elements are not merely some utility elements for staged languages, but could play a fundamental role in all cases where

source code needs to be manipulated. In particular, if supported with the required IDE extensions, they can substitute transformation languages by a combination of processes and libraries.

6.2.2 Aspect Transformation Library

As previously discussed, the metaprogramming elements of the language are sufficient for any AST transformation and thus for introducing cross-cutting functionality. However, we further facilitate the development of aspect programs by providing an AST transformation library with functionality that resembles the typical AOP style.

Since we target AST transformations, joinpoints essentially match specific AST node types to which advice functionality can be added. For example, we support the typical joinpoints like the call and execution of a function or method, the execution of an object constructor, the getting or setting of an object field and the execution of an exception handler. Each of them correspond to specific AST nodes; the function execution corresponds to the AST of the body of the matched function while the execution of an exception handler corresponds to the AST of the matched exception handler's body. Pointcuts are expressed as string literals and are matched against AST nodes using a custom pattern matching language. For example, the pointcut `method m(*)` will match nodes corresponding to method definitions with name `m` and any number of arguments. We support the typical pointcuts covering the basic joinpoints, pointcut combinators (i.e. `and`, `or`, `not`) for composition as well as some pointcuts specific for AST manipulation. For instance, the `ast(type)` pointcut matches all AST nodes that have the given type, the `parent(pattern, [childId])` matches parent nodes whose child at index `childId` (or any child if not specified) satisfies the given pattern while the `descendant(pattern)` matches the nodes that are part of a sub-tree whose root node is of the given type. Such pointcuts allow specifying fine-grained aspect transformations on a target AST. For example if we want to advise the `break` statements of a `for` loop within some method `m` we can use the following pointcut: `"ast(break) and descendant(for) and descendant(method m(*))"`. In the same sense, the pointcut `"ast(assign) and parent(id(x), lvalue)"` will match all assignments

whose child with index `lvalue`, i.e. whose left value, is an identifier `x`, for instance `x = 1`, `x = f()`, etc.

The main function of our library is `aspect(target:ast, pointcut:string, advice_type:enumerated, advice:ast)` that given a target AST and the pointcut to match will insert the advice AST as specified by the advice type. The target argument may specify either the entire program AST or any of its sub-trees that may have been obtained through custom AST traversal or prior node matching against some criteria. Regarding the advice type, we support *before*, *after* and *around* advice, meaning that the given code may be inserted respectively before, after or around the matched joinpoint. The exact way that advice code is inserted depends on the joinpoint and the matched AST node; for example, when we match the execution of a function, *before* advice inserts the given code at the beginning of the matched function body, while *after* advice inserts its code at all exit paths of the matched function body. For *after* advice applied on function or method execution in particular, we can use the delayed escape `<< ... ~~retval ... >>` that will carry the original return value of the function. Another delayed escape, specifically `<< ... ~~proceed ... >>`, is also typically used in *around* advice. The advice is applied by firstly substituting the AST being advised with the given advice AST and then by replacing the delayed escape (`~~proceed`) with the original AST value. For example, when applying the around advice `<<print("before"); ~~proceed; print("after")>>` on the AST of a function call `<< f() >>` the result will be `<<print("before"); f(); print("after")>>`. This construct can also be combined with ASTs that contain staging annotations. For example, the advice `<<!(~~proceed)>>` can transform the expression `f()` into `!(f())`, while the advice `<<~~(~~proceed)>>` can transform `x` into `~x` (the first delayed escape represents a single `~` while the `~~proceed` is typically replaced by the target AST, here `x`). Essentially, when using around advice, the last argument to the `aspect` function can be seen as a process that takes the matched AST and transforms it as described by the target AST.

To allow explicit transformation logic while still relying on pattern matching we also provide two additional functions: `match(target:ast, pointcut:string)`, that will find and return all nodes within the target AST that match the given pointcut

and `advice(target:ast, advice_type:enumerated, advice:ast)` that will insert the advice AST in the target as specified by the advice type. A summary of the basic elements offered by our AOP library is provided in Table 6.3.

Table 6.3 – Overview of the basic elements offered by our AOP library

Library functions	<i>aspect(target:ast, pointcut:string, advice_type:enumerated, advice:ast) : void</i>		
	<i>match(target:ast, pointcut:string) : list<ast></i>		
	<i>advice(target:ast, advice_type:enumerated, advice:ast) : void</i>		
Advice type	<i>BEFORE</i>	<i>AFTER</i>	<i>AROUND</i>
Basic Pointcuts	<i>execution(pattern)</i>	<i>call(pattern)</i>	<i>exception(pattern)</i>
	<i>class(pattern)</i>	<i>setter(field)</i>	<i>getter(field)</i>
AST Pointcuts	<i>ast(type)</i>	<i>child(pattern)</i>	<i>parent(pattern)</i>
	<i>descendant(pattern)</i>	<i>ascendant(pattern)</i>	<i>construction(pattern)</i>
Pointcut Combinators	<i>pointcut and pointcut</i>	<i>pointcut or pointcut</i>	<i>not pointcut</i>

6.2.3 Aspects in the Workspace Manager

In a system supporting binary-level weaving, the aspect sources are typically placed along with the normal program sources in the workspace management. For instance, in the *AJDT* [Eclipse05] eclipse plugin for AspectJ, there are aspect-enabled projects that can host both normal Java and AspectJ source files whose generated code is woven together after compilation. In a system with source-level weaving, the aspect transformation has to be in executable form while a normal program is still in source code waiting to be transformed before its compilation. This means that aspect sources and normal program sources are compiled at different times and thus should be properly distinguished in the workspace management. Particularly in our system, where aspects are implemented as typical programs within the same language and their separation with normal programs relies only on their different deployment, supporting such a distinction is even more critical.

Our system supports this distinction by introducing the notion of *aspect sources* that are organized in *aspect projects*. An aspect source contains all typical source information required for its build and deployment (e.g. compilation flags, dependencies, runtime libraries, etc.) as well as information about its transformation purpose, i.e. if it is a pre-staging, an in-staging or a post-staging aspect. This information is explicitly provided by the programmer who specifies the transformation category for each aspect source. In fact, for a single aspect it is

possible to specify more than one category, for instance both in-staging and post-staging. The reason for this is that stages may involve computations typically found in a normal program, so they may also require similar crosscutting functionality. Thus, a single aspect is allowed to address both stages and normal programs.

Aspect projects are used for grouping aspect sources and allow specifying the ordering of multiple aspect sources of the same type. For each project or source file within the workspace, the IDE allows specifying the aspect projects that will be used to advise it. As aspect programs are also programs, aspect sources inherit all properties of normal sources and they can be advised as well. This means that it is possible to use an aspect transformation to manipulate the code of another aspect transformation (but not of itself, as that would require a pre-existing binary of its code).

6.2.4 Aspects in the Build Process

Aspect transformation may be part of the compilation loop; however the aspect weaver need not be tightly coupled with the compiler. Actually, they may both be unaware of the existence of the other and let the build system orchestrate their interoperation.

The aspect weaver just takes an input source, transforms it one or more times and gives as output the resulting output source, thus naturally involving no additional interoperation with either the compiler or the build system. On the other hand, the compiler receives as input a source file and gives as output a binary file; however it requires interoperating with the build system to handle the build process of any stages involved in the process. In this sense, interaction between the build system, compiler and aspect weaver, illustrated in Figure 6.7, is as follows. When a source is to be built, the build system invokes the weaver with that source as input (step 1), applies the associated pre-staging aspects, receives its output (step 2) and then uses that as input to the compiler (step 3). Then, during the staging pipeline, the meta-compiler assembles the stage source (step 4) and asks the build system to build it (step 5) and provide its binary code; that code will then be execute to update the AST of the initial program being compiled. After receiving the stage source, the build system can invoke the aspect weaver to apply the in-staging aspects (step 6), get the transformed stage source (step 7) and send it for compilation on a new compiler instance (step 8).

The nested compilation will provide the stage binary (step 9) that the build system can then supply to the original compiler (step 10) to continue its stage execution. After the current stage execution, if there are still additional stages the same process is repeated (step 11). Eventually, there will be no more staging and the source code resulting from the staging process is ready to be built (step 12). This final source is then propagated to the weaver for applying the post-staging aspects (step 13), and the result is sent to yet another compiler instance (step 15) that will generate the final binary code (step 16).

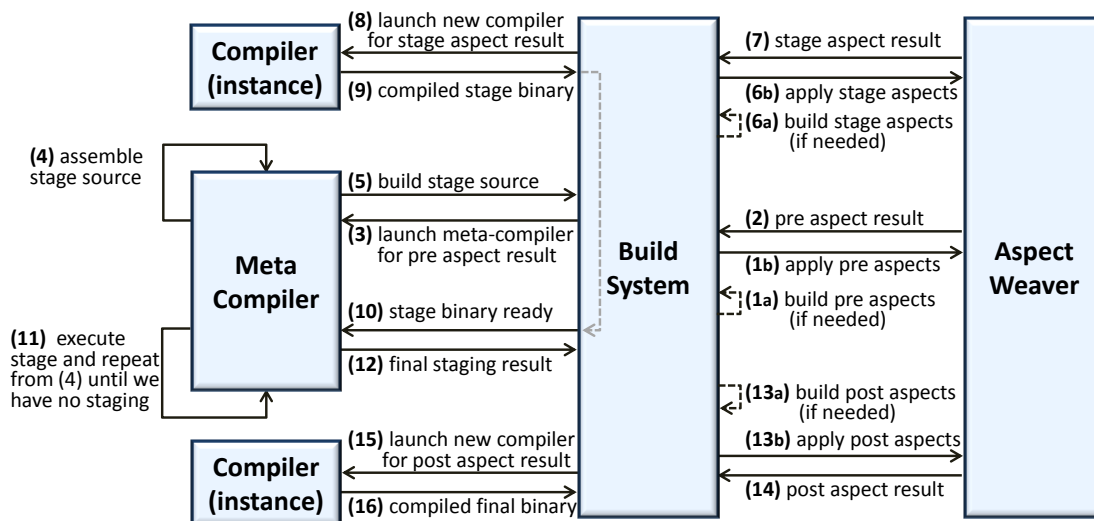


Figure 6.7 – Interaction sequence diagram between the build system, compiler and aspect weaver.

This description assumes that the aspect transformations are already available in binary form. In general, this may not be the case, so before applying any aspect transformations, the build system may first have to build them to get their binary form (Figure 6.7, steps 1a, 6a, 13a). Such build steps are automatically performed by the system if needed and may involve additional meta-compilation, in case the target aspect source contains meta-code, or even recursive aspect transformations, if the target aspect is also advised by another aspect. Essentially, a single build request for the initial source may trigger multiple nested build requests for metaprograms or aspect programs involved directly or indirectly in the process.

For instance, consider a pre-staging aspect source that contains meta-code and a normal source with no meta-code that will be advised by the aspect source. When trying to build the normal source we require the binary of the aspect source, meaning we have to build it first (normal source, step 1a). Since the pre-staging aspect involves

no aspect transformations of its own, it is directly sent to the meta-compiler that handles the staging process and returns a binary for it (aspect source, steps 3-5, 8-12 and 15-16). Then we can continue with the weaving of the normal source and the subsequent meta-compilation process that will result in the final binary (normal source, steps 1b-5, 8-12 and 15-16). Of course, if either the aspect source or the original source were advised by additional in-staging or post-staging aspects, there would be further aspect weaver invocations (i.e. steps 6-7 and 13-14) and possibly additional nested build requests. Overall, the build process is a recursive process that relies on the following principles:

- Building a source that is advised by specific aspect projects requires recursively building all aspect sources of these projects, invoking the aspect weaver to transform the initial source and then recursively building the last transformation result.
- Building a source with no aspect transformations, but containing meta-code requires assembling each stage, building it recursively, executing its binary code to update the main program AST and finally recursively building the final source when no more meta-code is present.
- Building a source with no aspect transformation or meta-code requires recursively building any module dependencies for the initial source and then finally typically translating its source code into binary.

6.2.5 Debugging Aspects

Utilizing aspects or metaprograms in a development process is a challenging task on its own. Trying to combine the two presents an inherently increased level of complexity, requiring the IDE to provide advanced tool support for writing and debugging programs in order to help programmers in this demanding task. In this direction, we extend our previous work on tool support for metaprogramming [Lilis12A], to also support aspect-oriented transformations. Since we build upon our implementation for aspect support, the discussion is focused on source-level weaving. Nevertheless, the feature implementation could also utilize binary-level weaving, while the rationale for their support is still valid in both cases.

6.2.5.1 Reviewing Woven Code

When aspect code is woven together with normal program code, either through source- or binary-level weaving, the result is a transformed version of the code that is not available to the programmer. This may not be an issue when the resulting code behaves as expected or the aspect is simple enough to verify its functionality in a few execution sessions. However, if the resulting code does not behave as expected or the aspect involves some complex pointcuts, information about the final version of the code can be invaluable to programmers, allowing them to see how the aspect application transformed the code and figure out the reason of the erroneous behavior. Reviewing the results of aspect weaving can provide helpful information even when the resulting code executes correctly, as it enables programmers to move from an abstract representation of the final code to a concrete visualization, increasing their understanding of the transformation that takes place.

This is a similar requirement with the reviewing of the updated version of the main program after having evaluated some stage metaprogram [Lilis12A]. As such, it is addressed in a similar way by *unparsing* the AST produced by the aspect transformation into source text, storing that text into a source file, and finally notifying the IDE to insert it in the workspace, properly associating it with the original source. Sparrow already supports this functionality for metaprogram results,

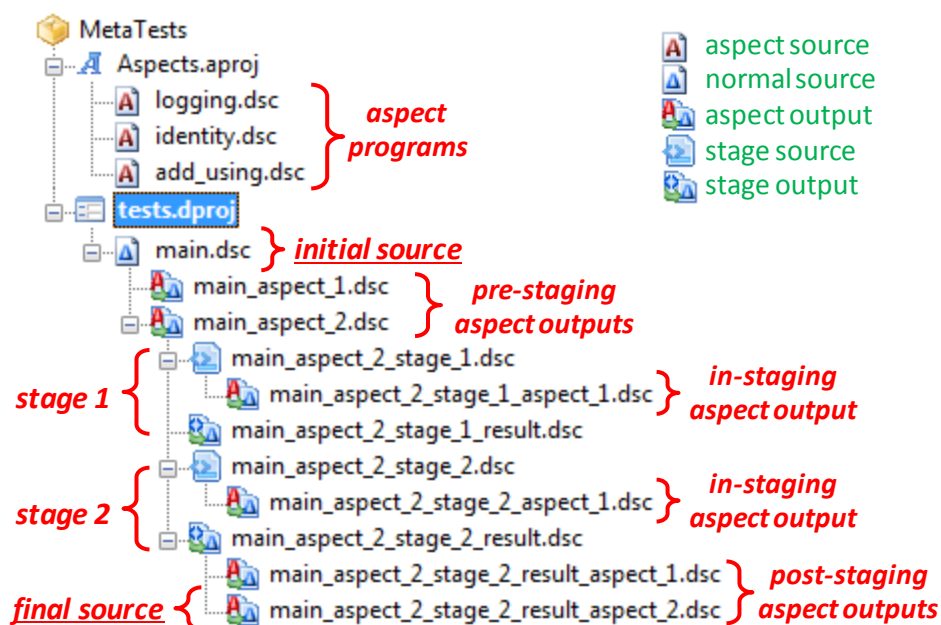


Figure 6.8 – Sample workspace showing source files generated by staging and aspect transformations.

so the only addition required involves the aspect weaver. In case of multiple aspects, the weaver generates an updated version of the source code after applying each separate transformation, thus providing a full trajectory of the transformation process. Figure 6.8 illustrates a sample workspace involving files generated by both staging and aspect transformations.

6.2.5.2 Providing Accurate Compile Errors

When the source code of a program that has passed through multiple transformation steps contains errors, it is not clear whether the error was present in the original source or if it was introduced as a result of one or more of the transformations [Tratt08]. In this sense, instead of a single error report specifying the final error location, the compiler should be able to track down and provide the first introduction of the erroneous code as well as the complete transformation chain that led to its final form. To achieve this functionality it is possible to associate any generated source location with the source location it originated from before the transformation took place. This way, we can create a list of source references that can track the error across all source files involved in the compilation.

A similar error tracking scenario is involved in the typical metaprogramming process requiring the creation and maintenance of a list of source references [Lilis12A]. However, in that case the entire process takes place within the compiler that simply provides the IDE with all relevant source reference information upon generating the error report. With the introduction of aspect transformations, we have a separate aspect weaver process responsible simply for source transformations and unaware of the source references maintained by the compiler. Since the aspect weaver and the compiler have to be as loosely-coupled as possible, the infrastructure for the source references has to be moved within the IDE, stored as metadata accompanying each generated source file. This way, whenever a generated source is created by a transformation process, either due to aspects or metaprogramming, we also have to provide the associated source references to the IDE. With this information, the IDE can then track down all sources involved in the generation or transformation of an error and form the entire transformation chain, properly associating it with any issued error report. This functionality is illustrated in Figure 6.9, where the given error report refers to all source files involved in the generation of the erroneous code. This allows

the programmer to navigate across the various source files versions (as discussed they are available in the workspace), reviewing the transformations performed from one version to the next and eventually understanding which transformation introduced the error.

```
Error, file 'main_aspect_1_stage_2_result_aspect_2.dsc', line 19:
Expression '(g)' not a callable value (its type is 'Undefined').
See file 'main_aspect_1_stage_2_result_aspect_1.dsc', line 18.
See file 'main_aspect_1_stage_2_result.dsc', line 12.
See file 'main_aspect_1_stage_2_aspect_1.dsc', line 12.
See file 'main_aspect_1_stage_2.dsc', line 12.
See file 'main_aspect_1_stage_1_result.dsc', line 12.
See file 'main_aspect_1_stage_1_aspect_1.dsc', line 32.
See file 'main_aspect_1_stage_1.dsc', line 32.
See file 'main_aspect_1.dsc', line 32.
See file 'main.dsc', line 11.
Finished compiling 'main_aspect_1_stage_2_result_aspect_2.dsc',
1 ERRORS detected.
```

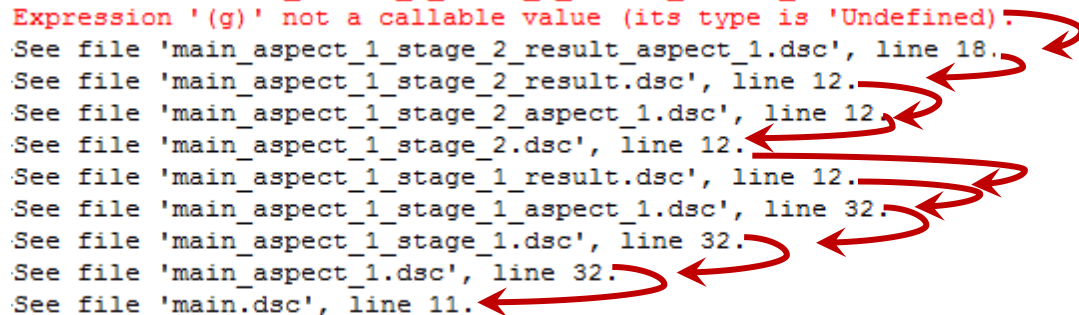


Figure 6.9 – Tracing compile errors in the source transformation pipeline involving staging and aspects; source names refer to the workspace of Figure 6.8.

6.2.5.3 Tracing the Evaluation of Aspects

Being able to view the result of an aspect transformation is a step closer to debugging aspect applications; however it lacks the information about *how* the code reached its final form. Such information involves tracing the entire control flow of the transformation logic as well as inspecting the transformation data. Essentially, the requirement is to provide full- fledged source-level debugging of the aspect program that is invoked to perform the transformation. Any aspect program is executed during the compilation process and performs AST modifications, so it resembles the execution of a normal compile-time metaprogram. In this sense, we can reuse the IDE debugger front-end functionality for compile-time debugging [Lilis12A] and instrument the aspect weaver with a debugger back-end that will handle the execution of the aspect programs. This way we can support debugging of the aspect transformation logic without practically making any changes to the existing infrastructure. An example of such a debugging session is illustrated in Figure 6.10.

As we mentioned earlier, in-staging aspects transform stage metaprograms before they are evaluated, so the two execute sequentially. From a debugger perspective, this means that the front-end has to be able to support multiple different back-ends.

Additionally, all stage executions take place within the compiler, meaning they are served by a single debugger back-end, while the aspect transformations for different stages are executed by different aspect weavers, meaning they are served by multiple debugger back-ends. Essentially this means that the debug session of the stage metaprogram can be interleaved with the debug sessions of the in-staging aspects.

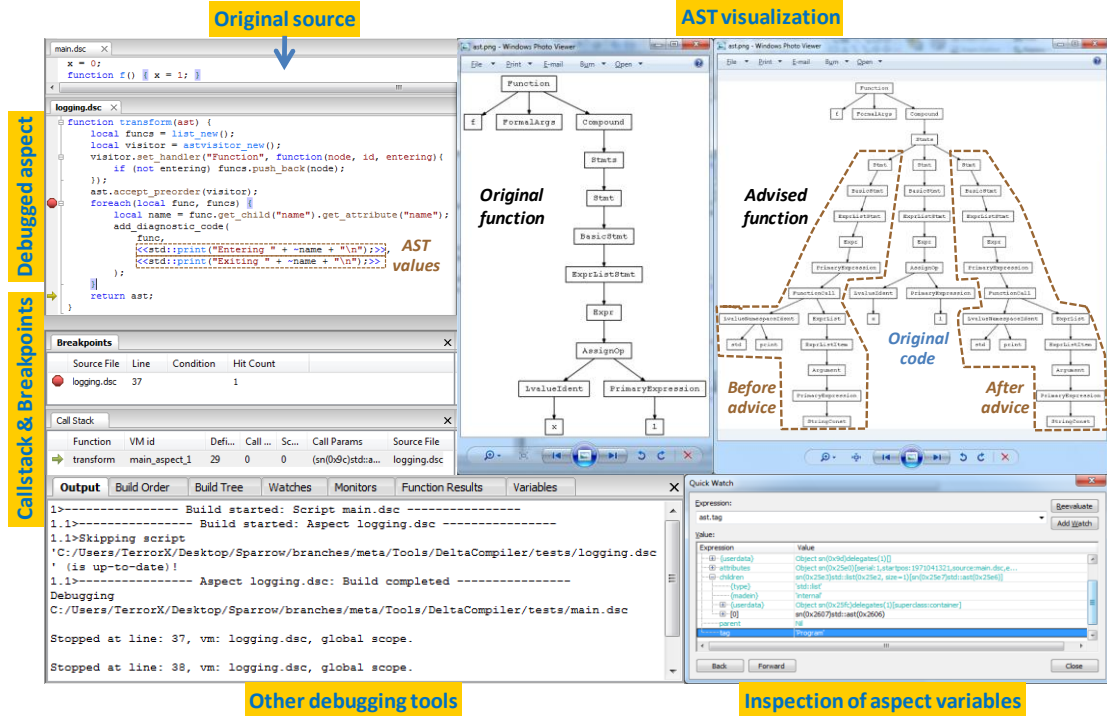


Figure 6.10 – Full-scale source-level debugging of aspect programs when they are actually applied during compilation.

For example consider a program with two stages of metaprograms where each of these stages is subject to an aspect transformation. When compilation begins, the debugger back-end within the compiler will connect with the IDE front-end. After the first stage is composed it will be sent to the aspect weaver for transformation. Upon launching the weaver, its debugger back-end will also connect to the IDE front-end overriding the previous connection. The weaver will then proceed with the execution of the aspect transformation that is the first program that will be debugged. After finishing the transformation, the debugger back-end of the weaver disconnects from the IDE front-end, restoring the compiler debugger back-end as the active one. The compiler then proceeds with the execution of the first stage that is the second program that is debugged. When the stage execution is finished, the initial program AST is transformed and ready to compose the second stage. The same process is repeated,

with the launch of the aspect weaver creating a new active debugging connection for the aspect transformation of the second stage and its termination restoring the compiler’s debugging connection as the active one, to finally debug second stage execution.

Offering the functionality described above requires extensions in the infrastructure; in particular allowing the debugger front-end to handle multiple back-ends. However, considering the execution order of the systems involved in the process, namely the compiler and the instances of the aspect weaver, it is clear that no two systems may run in parallel. This means that the only required extension is to support adding and removing debugger connections, while typically serving the one added most recently.

6.3 Case Studies

We present various case studies illustrating the potential benefits from incorporating aspects in the staging pipeline. In particular, we focus on scenarios concerning pre-staging and in-staging aspects. We do not provide examples explicitly post-staging aspect as any typical AOP aspect is also a valid post-staging aspect.

6.3.1 Aspects to Insert Staging

In [Taha04], Taha describes a methodology for taking conventional programs and turning them into multi-stage programs thus reducing potential runtime overhead and improving performance. For instance consider the classic *power* example.

```
function power (x, n) {      ← original power version
  if (n == 0) return 1;
  else return x * power(x, n - 1);
}
a = 2; print(power(a, 4));   ← recursive invocation at runtime

function spower (x, n) {    ← staged power version
  if (n == 0) return <<1>>;
  else return <<~x * ~(spower(x, n - 1))>>;
}
a = 2; print(!(spower(<<a>>, 4))); ← final code is: print(a*a*a*a*1);
```

Typically, the staged version has to be explicitly written by the programmer. However, it is possible to turn the methodology into an algorithm allowing the automation of this process (i.e. a methodology for transforming function `power` into `spower` automatically). Implementing such an algorithm requires analyzing the AST of the target program to locate potential for deploying staging and then transforming it

appropriately to introduce the necessary staging annotations. In this sense, the application of the algorithm can be seen as an aspect-oriented transformation that weaves the advice functionality, i.e. staging annotations, at the desired pointcuts, i.e. source locations with staging potential. In particular, this is a *pre-staging aspect*; it will transform the initial source, originally containing only normal code, to enrich it with staging annotations. The aspect is shown below with `transform` being its entry point. Notice the use of the previously described AST-wise pointcuts like *descendent*, *child*, *ast*, etc. used for fine-grained AST matching.

```

toAST = << << ~proceed >> >>;    ← transforms x → <<x>>, used below
addEscape = <<~( ~proceed)>>;    ← transforms x → ~x, used below
addStaging = <<&~proceed>>;    ← transforms func f(){} → &func f(){} , used below
addInline = <<!( ~proceed)>>;    ← transforms f() → !(f()), used below
...the above are used for around advice, applying the corresponding transformations...
function InFunc(name) { return "descendant(function "+name+"(..)"; }
function MatchCall(name) { return "call(" + name + "(..)"; }
function OutsideRecursiveCall(funcName)
{ return "not descendant(" + MatchCall(funcName) + ")"; }
...the above are helper functions to create pointcut expressions...
function StageDefinition(func) {    ← turns the function body to a staged form
  local recursiveCall = MatchCall(func.GetName());
  aspect(func, recursiveCall, AROUND, addEscape);    ←escape recursive calls
  local exprs = match(func, "child(return)");    ←begin with all return exprs
  while(not exprs.empty()) {    ← until all exprs for the result are handled
    local dependencies = list_new();    ← holds the deps for the current exprs
    foreach(local expr, exprs) {
      advise(expr, AROUND, toAST);    ← turn expr into AST form, i.e. <<expr>>
      ids=match(expr, "id(*) and not descendant("+recursiveCall+"");
      foreach(local id, ids) {    ← for all matched ids (args&locals) in expr
        dependencies.push_back(id.GetName());    ← mark id as a dep
        advise(id, AROUND, addEscape);    ← escape the id
      }
    }
    exprs.clear();
    foreach(local x, dependencies) {    ← check for assignments to deps
      assigns=match(func, "ast(assign) and parent(id("+x+"), lvalue)");
      foreach(local assign, assigns)    ←for all matched assigns recursively
        exprs.push_back(assign.GetChild("rvalue"));    ←check the rvalues
    }
  }
  advise(func.GetParent(), AROUND, addStaging);    ←stage entire function definition
}

function StageCalls(ast, funcName) {
  calls=match(ast, MatchCall(funcName)+"and not "+InFunc(funcName));
  foreach(local call, calls) {    ← for each matched non-recursive call
    foreach(local actual, call.GetActuals())    ← iterate over actuals
      if(not actual.IsConst())
        advise(actual, AROUND, toAST);    ← stage args
        advise(call, AROUND, addInline);    ← stage entire call with inline tag
  }
}

```

```

function transform (ast) { ← ast holds the code to be transformed
  foreach(local func, match(ast, "function *(..)")) ← find all functions
    if (CanBeStaged(func)) { ← check if the result can be expressed as a
      ← mathematical expression over input arguments
      StageDefinition(func); ← stage the function definition
      StageCalls(ast, func.GetName()); ← stage calls across the entire ast
    }
  return ast; ← the transformed ast is the result of the aspect weaving
}

```

In particular, the aspect will first try to find functions that have potential for staging. Without going into details, this process essentially looks for functions whose result can be expressed in a mathematic expression over their input arguments. Power, as well as other mathematical functions like factorial, fibonacci, etc., fit the above description and will be matched by the aspect. For each of the matched functions, we need to stage both definition and invocations. For the definition, we have to stage all items relating to the function result. In this sense, we begin by properly staging the return expressions of the function while marking any argument or local variables involved in their computation. We then repeat the same process targeting any assignment to the previously marked variables. We properly stage the right hand side of each assignment and mark any additional arguments or local variables involved in its computation. This process continues iteratively until all involved variables have been handled. We should also note that any recursive function invocations are by default considered to be involved in the final function result so they are staged upfront and then excluded from the remaining process (hence the recursive call pointcut). For the power example in particular, this process will transform `return 1;` to `return <<1>>;` and `return x * power(x, n - 1);` to `return <<~x * ~(power(x, n - 1))>>;`. This is achieved by applying around advice and specifying `toAST = << << ~~proceed >> >>` and `addEscape = <<~~(~~proceed)>>` as advice targets. The former essentially turns `1` into `<<1>>` and `x * power(x, n - 1)` into `<<x * power(x, n - 1)>>` while the latter further transforms `<<x * power(x, n - 1)>>` into `<<~x * ~(power(x, n - 1))>>`.

Then, for each invocation, the aspect will introduce the *inline* operator and turn any non-constant argument to its corresponding AST form. In the power example, this process will transform the invocation `power(a, 4);` toAST will turn `a` into `<<a>>`, while `<<!(~~proceed)>>` will further transform `power(<<a>>, 4)`

into `!(power(<<a>>, 4))`. The result is essentially the automatic staging of all relevant function invocations that achieves the desired performance gain. This would not have been possible without the pre-stage aspect, as the original program contained no staged code and its compilation would yield binary code where all functions and their invocations maintained their original form.

This may not be a representative AOP example, but it shows how a pre-staging aspect should operate, i.e. updating or changing the staging of a program, and illustrates a scenario where such functionality is useful. In fact, this example relates to partial evaluation and would be typically handled by a partial evaluator without requiring the extra aspect specification. However, the binding-time analysis involved in partial evaluation is not complete and can only approximate the knowledge of the programmer, meaning that explicitly specifying how the code should be staged may yield better results. Additionally, implementing the aspect involves mainly AST manipulation that a programmer is familiar with, while effective use of a partial evaluator involves a steep learning curve [Jones].

Considering the specific power example, writing and applying the aspect is of course more complicated than staging the code manually. However, the aspect is generic enough that it can be used for a variety of other mathematical functions without the need for manually staging each of them. Additionally, the aspect automatically locates and stages all function invocations; without that, the programmer would have to locate all such invocations (probably multiple ones, scattered in the source code) and stage them manually.

6.3.2 Aspects for Custom Static Analysis

During the compilation process, a compiler typically performs a series of static analysis checks to the program being compiled. However, a programmer is typically not aware of the checks being performed while also being unable to customize their behavior. The latter can be achieved by placing staged code at specific locations within the original source so that their execution performs the desired static analysis checks. In this direction, we can use a *pre-staging aspect* to introduce the custom analysis code along with its deployment. For example, the aspect below introduces staged code to analyze all functions definitions.

```

function transform (ast) {
  foreach(local func, match(ast, "ast(function)"))  ←match all functions
    advise(
      func,
      AFTER,
      <<&analyze(compiler::get_function_ast(~(func.GetName()))>>;
    );
    advise(ast, BEFORE, <<&function analyze(func) {...}>>; ←insert staged def
  return ast;
}

```

6.3.3 Aspects to Introduce Memoization in Stages

To improve runtime performance for mathematical functions involving intense computations a common technique is to generate for them constant tables, i.e. tables that will map specific function arguments directly to a constant value. Such tables can be generated by metaprograms; for example, consider the following code that generates a constant table for a range of Fibonacci numbers. As indicated by the `&` annotation, functions `fibonacci` and `GenerateFibonacciTable` are staged; in particular, the latter uses the former to calculate the required values and merge them into a constant table that is inlined in the program code (`!(GenerateFibonacciTable(20))` invocation). At runtime, function `fib` will provide the result directly by accessing the generated constant table.

```

&function fibonacci(n) {  ← compile time version using normal computation
  if (n == 0 or n == 1) return 1;
  else return fibonacci(n - 1) + fibonacci(n - 2);
}
&function GenerateFibonacciTable(upperBound) {
  local numbers = nil;
  for (local i = 0; i < upperBound; ++i)
    numbers = <<~numbers, ~(fibonacci(i))>>;  ←merge computed values
  return << [~numbers] >>;  ←generate const table with the resulting values
}
function fib(n) {  ← runtime version using the generated const table
  static table = !(GenerateFibonacciTable(20)); ←inline the const table here
  return table[n];
}
print("fib(15) = ", fib(15));  ← call involves no runtime overhead

```

While the above technique improves *runtime performance*, during compilation the metaprogram still has to compute the required values, something that may take a long time. To improve *compile-time performance* (metaprogram execution), we can use *memoization*, i.e. caching the result of a function to avoid recalculating it with the same arguments. This functionality is not coupled to a specific metaprogram but would apply to any metaprogram with similar functionality. As such, it can be

expressed as an *in-staging aspect* that will be used for each such metaprogram advising its function invocations with memoization. The *fibonacci* example above can be advised with memoization by the following aspect code:

```
function transform (ast) {
  local pointcut = "execution(function fibonacci(n))";
  local beforeAdvice = <<
    static memoizer = [];    ←memoization cache
    if (memoizer[n] != nil) return memoizer[n];
  >>;
  local afterAdvice = <<memoizer[n] = ~~retval;>>;
  ...the above advice memoizes the result of the calculation; ~~retval carries the return value...
  aspect(ast, pointcut, AFTER, afterAdvice);
  ...put the before advice second to avoid advising the return present in it...
  aspect(ast, pointcut, BEFORE, beforeAdvice);
  return ast;
}
```

6.3.4 Aspects for Tracing Diagnostics in Stages

Stages may contain code that was never part of the original program and thus it may not be easy to trace their execution when they don't behave as expected. In such cases, unless the IDE provides support for debugging metaprograms, the only option is to manually insert logging calls within functions of the stage program to trace their execution. However, logging is a well-known crosscutting concern that can be addressed through AOP. In this sense, using the following code as an *in-staging aspect* achieves the desired functionality.

```
function transform (ast) {
  local funcs = match(ast, "execution(function *(*))");
  foreach(local f, funcs) { ←iterate over all matched function definitions
    local name = f.GetName();
    advise(f, BEFORE, <<print("Entering " + ~name);>>);
    advise(f, AFTER, <<print("Exiting " + ~name);>>);
  }
  return ast;
}
```

Note that since we use an *in-staging aspect*, the `ast` argument passed to the `transform` function will only contain stage code. As such, the tracing functionality is only introduced in functions available within stages, and not the functions of the final program.

6.3.5 Aspects for Locking Shared Objects in Stages

Since stages are normal programs, their execution may involve multiple threads of execution that share various resources. This raises the issue of protecting the stage

code from possible race conditions by introducing typical synchronization constructs like mutexes. This can be achieved using a locking aspect as illustrated in the following code.

```
function transform (ast) {
  local class = "class(SharedObject)";           ← class for synchronization
  local pointcut="descendant("+class+") and execution(method *(..))";
  aspect(ast, pointcut, BEFORE, <<self.mutex.lock();>>);
  aspect(ast, pointcut, AFTER, <<self.mutex.unlock();>>);
  aspect(ast, class, BEFORE, <<@mutex:mutex_new();>>); ← insert mutex member
  return ast;
}
```

6.3.6 Aspects for Exception Handling in Stages

As already discussed, the code of a stage metaprogram may be sophisticated and involve multiple scenarios where errors can occur. In this context it is a typical practice to use exception handling to separate the normal execution from the error handling code. Exceptions can be seen as a crosscutting concern allowing them to be modularized as aspects [Kiczales97]. In this sense, stage code could utilize an *in-staging aspect* to be advised with the error handling logic. For example, the following aspect can be used to specify different exception handling policies for a variety of use cases.

```
function AllMethodsInClass(class)           ← helper to create pointcut expressions
{return "execution(method *(..)) and descendant(class("+class+"))";}
function transform (ast) {
  aspect(ast, AllMethodsInClass("RemoteObject"), AROUND,
    <<try { ~~proceed; } catch Exception { log(Exception); }>>
  );
  ← log and ignore any exception regarding remote object invocations
  aspect(ast, AllMethodsInClass("StackWithDbyC"), AROUND,
    <<try { ~~proceed; } catch ContractException { assert false; }>>
  );
  ← ensure no contract exceptions thrown by a class with Design by Contract
  aspect(ast, AllMethodsInClass("ConfigurationManager"), AROUND,
    <<try { ~~proceed; } catch IOException
      { throw [@class:"ConfigException", @source:IOException]; }>>
  );
  ← hide low level IOExceptions and raise higher level ones
  return ast;
}
```

6.3.7 Aspects for Decorating Classes in Stages

Stages are mainly code generators and thus they make extensive use of AST creation and manipulation. Even if the AST library offered by the language facilitates AST traversal and manipulation, programmers may still want to decorate AST values with custom functionality. To do so, one would have to implement an additional library and manually decorate AST creation occurrences in the code. The latter can be seen

as a crosscutting concern that can be addressed through the following *in-staging aspect*. In particular, the aspect will locate all quasi-quotes nodes (i.e. AST creations) and apply the desired decoration based on the language element they contain. Of course, any *inlines* and *escapes* have to be advised as well to retrieve the original AST value from the decorated object.

```
function transform (ast) {
  local quasiquotes = match(ast, "ast(quasiquote)"); ←find all AST creations
  foreach(local quote, quasiquotes) {
    if (quote.GetChild().GetType() == "class")
      advise(quote, AROUND, <<[
        @ast : ~~proceed, ←AST creations are replaced with objects
        method GetMethods () {...}, ← the original AST is stored as normal data
        method GetAttributes () {...}, ← custom methods added
        method BaseClasses () {...}
      ]>>);
    else if (quote.GetChild().GetType() == "function")
      advise(quote, AROUND, <<[
        @ast : ~~proceed, ←AST creations are replaced with objects
        @ast : ~~proceed, ← the original AST is stored as normal data
        method GetName () {...}, ← custom methods added
        method GetActual(n) {...},
        method GetLocals() {...}
      ]>>);
    else ...perform similar handling for other quoted language elements...
  }
  aspect(ast, "child(escape)", AROUND, <<~~proceed.ast>>); ←get original AST
  aspect(ast, "child(inline)", AROUND, <<~~proceed.ast>>); ←get original AST
  return ast;
}
```

6.3.8 Aspects for Custom AST Iteration in Stages

It is typical for stage code to traverse the tree structure of an AST value. In Delta, this is achieved through an AST visitor, where node types are associated with handler functions. For example, the following code will traverse the AST shown and invoke the associated handler (i.e. the anonymous function) for every function node contained within the AST.

```
ast = << function f() { return << function g() {} >>; } >>;
visitor = astvisitor_new();
visitor.set_handler("function", function(node, id, entering){ ... });
ast.accept_preorder(visitor);
```

The visitor does not differentiate between functions being directly within the traversed AST or inside any nested quasi-quotes it contains, meaning that in this example the handler will be triggered by both functions *f* and *g*. However, it is very common for the traversal to target only functions directly within the AST and not nested ones. To achieve this, we have to introduce additional handlers to keep track

of the stage nesting and modify existing ones to utilize this information. This can be modeled with the following *in-staging* aspect:

```
function transform(ast) {
  local visitors = "ast(assign) and" +
    "parent(call(astvisitor_new()), rvalue)";
  foreach(local visitor, match(ast, visitors)) {
    local handlers = "execution(function (node,id,entering)) and " +
      "descendant(call("+id.GetName()+".set_handler(..))"; ←match handlers
    aspect(visitor.GetEnclosingBlock(), handlers, AROUND,
      << if(nesting==0) ~~proceed; >>);
    local id = visitor.GetChild("lvalue").copy();
    advise(visitor.GetEnclosingStmt(), AFTER, <<
      local nesting=0; ←introduce nesting var and modify it as needed in following handlers
      ~id.set_handler("quasiquotes", function(node, id, entering)
        {if(entering) ++nesting; else --nesting;}); ←increased within quotes
      ~id.set_handler("escape", function(node, id, entering)
        {if(entering) --nesting; else ++nesting;}); ←decreased within escapes
      >>);
  }
  return ast;
}
```

Notice that we first update existing handlers and then introduce the new ones so as to avoid advising them or having to specify a more complex pointcut that excludes them.

6.3.9 Aspects for AST Validation in Stages

ASTs are usually constructed through *quasi-quotes*, however they cannot express structures depending on some computation, for example having an *if* statement with a variable number of *else if* clauses. To allow generating such code patterns, metalanguages typically provide some extra facility, like a library for explicit AST creation and manipulation. ASTs created using either the library or through quasi-quotes should interoperate; however while ASTs created by quasi-quotes are well-formed, ASTs created through the library may be incomplete or even ill-formed. In this context, a programmer could insert custom validation code at specific source locations, ensuring that any AST is well-formed and that any manually constructed erroneous AST is reported as early as possible. This functionality can be achieved through an *in-staging aspect* that introduces a `validate` function available in stage code and weaves appropriate invocations to any source locations involving ASTs. In particular, this requires advising AST nodes corresponding to e. quasi-quotes, escapes and inlines. Additionally, we can advise any stage function operating on ASTs to also deploy a validation call for its argument. The source code for this in-staging aspect is provided below.


```

function transform(ast) {
  local validator = << function validate(ast) { ... return ast; } >>;
  ...the two following lines turn <<... ~x ...>> into validate(<<... ~(validate(x)) ...>>)...
  aspect(ast, "ast(quasiquotes)", AROUND, <<validate(~~proceed)>>);
  aspect(ast, "child(escape)", AROUND, <<validate(~~proceed)>>);
  ...the following line turns !(...) into !(validate(...))...
  aspect(ast, "child(inline)", AROUND, <<validate(~~proceed)>>);
  aspect(ast, "execution(function *(ast,...))", BEFORE,
    <<validate(ast);>>); ← also validate any function with an AST as first argument
  ...insert the validate func last to avoid advising it or having to specify a more complex pointcut...
  advise(ast, BEFORE, validator);
  return ast;
}

```

6.4 Discussion

We continue by discussing some elements that may differ in other languages and provide an overview for deploying our approach using a mainstream AOP language like AspectJ.

Scope extrusion In the Delta language, variables within quasi-quotes are typically dynamically scoped in the context where the quoted code will actually be inserted. In this sense, there are no guarantees regarding name bindings and as such no scope extrusion issue. However, our proposition towards AOP for stages is orthogonal to such an issue. In a language where symbols within quasi-quotes bind to specific variables via lexical scoping, the same language facilities that are used to guarantee the name binding for normal program compilation can be extended to also apply for any aspect transformations. For example, Template Haskell [Sheard02] and Converge both use the notion of *original names* to bind quasi-quoted symbols to top-level definitions within a module. In particular, for a top-level function f within a module M , any reference to f used within quasi-quotes is directly translated to $M:f$ ($M.f$ for the Converge version), uniquely referring to the particular name. In the same sense, any quasi-quote of an aspect transformation could also refer to the same function f using an extension for original names. Since the name of the module is not directly available during the compilation of the aspect program, we could instead use a special delayed escape `~~module` that will be replaced with the name of the module when it becomes available. This way, we could directly use original names within quasi-quotes, for instance writing `<<~~module:f>>`.

Interaction and Commutation Among Aspects Our implementation realizes aspects as separate AST transformation programs that are applied sequentially. In this

sequence, any aspect being applied operates on the result of earlier aspects, so naturally the application order is important; in fact, applying aspect programs with a different order may yield different results meaning that this is not a commutative process. For example, consider an aspect for introducing additional members to a class and another one for automatically generating accessor functions for the class members. If the former aspect is applied first the resulting class will have accessor functions for all members while if it is applied second any newly introduced members will have no accessor functions. Apart from the ordering issue, aspect transformations are applied once and for all, without the ability to be triggered again by other aspects. Essentially, an aspect may inspect changes introduced by earlier aspects, but not vice versa, effectively disallowing any bidirectional interaction between two aspects. Both limitations arise from the particular aspect implementation as transformation programs and are not inherent issues of our proposition for aspects in stages. In this sense, utilizing a more traditional AOP approach, with a separate aspect language and collective weaving of the aspect code along with the normal program or metaprogram code, aspects of the same category can interact with each other, while their commutation is the same as with normal aspects.

AOP for stages using Mint and AspectJ To apply our methodology using Mint and AspectJ, AspectJ first has to be extended to support the staging extensions of Mint. The latter is required so as to allow the aspect code contain staging annotations. Additionally, the stage binaries produced by Mint need to be available before they are executed so as to be advised by the aspect weaver. Essentially, the translation-execution loop required for the staging process has to provide an entry point allowing updating the original stage binary with the advised one. With these extensions, we can then follow the binary weaving shown in Figure 6.4. Initially, the original program is compiled to binary and is advised by the pre-staging aspects. As such, the program execution that follows uses the advised version of the binary. At runtime, whenever a stage binary is produced, the aspect weaver can intercept it, advise it with the in-staging aspects and then send it for execution. This way, the stage execution contains both original and aspect functionality. Regarding post-staging aspects, Mint uses runtime metaprogramming so, as previously discussed, they cannot be supported.

6.5 Comparison to Current AOP Practices

To our knowledge, this is the first work with a systematic proposition towards supporting aspects for stage programs in the context of multi-stage languages. However, we consider our work to be closely related to the attempts of using metaprogramming features for achieving aspect-orientation. For example, *AOP++* [Yao] is a generic AOP framework in C++ that utilized the metaprogramming constructs of the language, i.e. templates, to express pointcut expressions and match joinpoints at compile-time. *Nemerle* [Skalski04] facilitates metaprogramming through its macro system and can support AOP features by applying annotation based macro invocations on program classes. *AspectR* [Bryant] is a library for Ruby that utilizes metaprogramming techniques to implement AOP by wrapping code around existing methods in classes. *Groovy AOP* [Kaewkasi] is an AOP system for Groovy that provides a hybrid dynamic AOP implementation based on both metaprogramming and byte-code transformation. Aspects, pointcuts and advice are specified at compile-time based on a Groovy based domain-specific language while the advice is woven into byte-code at runtime using dynamic compilation.

Languages like Lisp or Scheme have a built-in notion of stages, while they also facilitate AOP through library support, for example using *AspectL* [Costanza] or *AspectScheme* [Dutchyn] respectively, thus allowing potentially combining stages and aspects. However, these libraries target generic AOP and do not provide explicit support for introducing aspects in staged code. Essentially this means that while from an expressiveness point of view it is possible to specify aspects for stages, from a software engineering point of view it requires introducing additional sophisticated macros, something difficult even for advanced users. The latter could be easily addressed with a dedicated AOP library for stages offering such macros out of the box and thus facilitating the adoption of AOP practices in staged code. It also shows that even languages with both concepts require a more systematic approach for their combined deployment.

MorphJ [Huang08][Huang11] is a language that introduces a form of metaprogramming by enabling the specification of general classes that are produced by iterating over members of other classes. In this sense, it can also be used to achieve AOP functionality by advising structural program features (e.g. before-, after-, and

around-advice for methods). As a program generation or transformation approach, MorphJ only allows enhancement of classes through subtyping or delegation. On the contrary, our system allows arbitrary code generation or transformation making it more expressive. As an AOP approach, MorphJ allows advising normal program code but cannot support advising its reflective transformation functionality, i.e. the metaprogram specifying the general class generation. A fundamental point of our proposition is that metaprograms may also require AOP functionality, so we support all stages of a multi-stage program to be subject to AOP. The advantage of MorphJ over our system (or other AOP tools) is the guarantee of modular type safety enabling the general classes to be type-checked independently of their uses. Indeed, in our system it is possible for an aspect program to be valid on its own, but cause errors upon its deployment. However, such an error is still reported during compilation, while the offered error reporting facility discussed in section 6.2.5.2 allows it to be easily identified and thus resolved.

There are also systems that provide dynamic AOP support through meta-object protocols or byte-code modification at load- or run-time. Examples include but are not limited to *JAC* [Pawlak], *Handi-Wrap* [Baker] and *Spring* [Johnson] for Java, *AspectS* [Hirschfeld] for Smalltalk and *AspectLua* [Cacho] for Lua. This approach is orthogonal to our work that focuses on systems with static AOP (like AspectJ).

Existing tools for debugging code involving AOP are also relevant to our work. For example, [Eaddy] and [Yin] offer support for debugging the final woven code while properly associating execution with the original source code or the aspect source code. Our system relies on source-level weaving and keeps the results of each aspect transformation, so source-level debugging of the woven code is straightforward by using the result of the final aspect transformation. Instead we focus on providing source-level debugging support for the *transformation logic*, allowing programmers view normal and aspect related code as ASTs and trace the entire weaving process.

Chapter 7

Advanced Practices

"Good programmers know what to write.

Great ones know what to rewrite (and reuse)"

-- Eric S. Raymond

A primary focus of this thesis was to derive a code of practice that will utilize metaprogramming techniques to achieve reusability at a macroscopic scale. In this context, we discuss three promising directions where metaprogramming can make a difference in the software development process: (i) implementing reusable design patterns by utilizing metaprograms as pattern generators; (ii) implementing reusable exception handling templates by adopting metaprogramming to express handler logic as parameterized source code fragments; and (iii) facilitating source code automation by encapsulating model-driven code generators as metaprograms. We continue by elaborating on the design rationale and proposed methodologies for each of the explored direction.

7.1 Design Pattern Generators

In software engineering, *design patterns* [Gamma] constitute generic reusable solutions to commonly recurring problems within a given context in software design. Effective software design requires considering issues that may not become visible until later in the implementation and design patterns can help preventing such problems by providing tested, proven development paradigms. A design pattern is not a complete design directly transformable into code; it is rather a description on how to solve the given problem in different situations illustrating relationships and interactions between classes and objects involved. This means that in general, a pattern has to be re-implemented from scratch each time it is deployed, thus emphasizing design reuse as opposed to source code reuse.

This issue was first identified in the attempt to turn patterns into context-independent reusable components [Arnout] without requiring developers re-implement the same

boilerplate code in every different context. The result was a classification of design patterns being componentizable or not, with the actual pattern component implementations where applicable. However, the approach was based on the Eiffel language [Meyer92] strongly relying on its *Agent* and *Design by Contract* mechanisms; thus it may not directly apply to other languages that offer no such features. Nevertheless, the idea of turning patterns into components has been explored in other languages as well. For example, *PerfectJPattern* [Garcia] is a framework and catalog of componentized design patterns for Java. It offers support for various design patterns, including some (e.g. *Adapter*, *Decorator*) that were classified as non-componentizable by Arnout. To deliver such functionality, the *PerfectJPattern* framework makes extended use of the Java reflection API, i.e. it relies on runtime metaprogramming.

Similar work has also been carried out in the context of compile-time metaprogramming. Nemerle supports generating the implementation details of a design pattern through meta-attribute based macros [Skalski05]. In particular, it offers implementations for the *Composite*, *Proxy*, *Singleton* and *Abstract Factory* patterns. Groovy AST transformations can also offer such functionality; in particular, the *@Delegate* and *@Singleton* annotations introduced in version 1.6 show the potential for automatically generating design pattern implementations.

Our approach also targets compile-time metaprogramming and shares the philosophy of expressing the pattern logic as a metaprogram while passing the particular application context as deployment parameters. In this sense, our contribution is not the adoption of metaprogramming for pattern generators per se, but our proposition for utilizing the integrated metaprogramming model to do so. Essentially, as discussed in section 4.5.2, to effectively accommodate the requirements for implementing design pattern generators requires features beyond staged expressions. With integrated metaprograms, programmers may apply normal program practices like encapsulation, abstraction and separation of concerns. Additionally, the support for shared state and typical control-flow enables the delivery of more elaborate pattern implementations involving matching modifications across multiple classes or the orchestrated insertion of specific pattern implementation details at disparate source locations.

We continue by discussing some examples of design pattern generators that utilizing our programming model. Our examples include patterns that were classified as non-componentizable by Arnout (e.g. decorator, adapter, singleton, etc.) illustrating that they can in fact be modularized and reused not as components but as parameterized source code fragments deployable through metaprogramming. Our work has been carried out in the context of the untyped Delta language; nevertheless it can be directly applied in a typed language. To show this we also discuss corresponding example implementations in a hypothetical meta-C++ that adopts our staging annotations and programming model.

7.1.1 Decorator

The *Decorator* pattern allows attaching additional responsibilities to an object dynamically. It targets individual object and not entire classes, so it provides a flexible alternative to subclassing for extending functionality. The basic idea is to enclose the target component in another object that provides the extended functionality, called *decorator*. The decorator forwards any requests to the target component while performing additional actions before or after that. It conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. This transparency allows decorators to be nested, thus supporting an unlimited number of added responsibilities.

The software engineering components involved in the Decorator pattern include the interface of the decorated object, an interface for the decorator class (conforming to the latter) and the concrete decorator classes. The latter are the main part required to deploy the pattern, involving only the class names along with the methods with refined functionality as the varying behavior that can be parameterized. In this sense, we can utilize a meta-function that will generate decorator classes for a target class based on the refined methods given as parameters. This is illustrated in the following example.

```
function Car() {                                     ←normal class for which we want to generate a decorator
  return [
    method Move      () {...},
    method Break     () {...},
    method Accelerate() {...},
  ];
}
```

```

&function DecoratorGenerator (class, decoratorSpecs) { ←pattern generator
  local allMethods = class.GetMethods();
  local decorators = nil;
  foreach(local spec, decoratorSpecs) {
    local delegations = nil;
    local refinedMethods = spec.refined.GetMethods();
    foreach(local name, allMethods)
      if (not refinedMethods.contains(name)) ←only for non-refined methods
        delegations = <<
          ~delegations, ← merge with any previous delegation methods
          method ~name(...) {@instance.~name(...);} ←add delegation method
        >>; ←... in formal means any formal, ... in call means pass all supplied arguments
    local decorator = << ← create new decorator class using a constructor function
      function ~(spec.name) (instance) { ←instance is the decorated object
        return [
          @instance : instance,
          ~delegations, ←class contains the non-refined delegation methods
          ~(spec.refines) ←along with the refined ones
        ];
      }
    >>;
    decorators = <<~decorators, ~decorator>>; ← merge decorator classes
  }
  return decorators; ← return a single AST containing all decorator classes
}

```

<pre> ! (DecoratorGenerator(GetClassDef("Car"), [[@name : "ABSCar", @refines : << method Break() {...} >>], [@name : "TurboCar", @refines : << method Move() {...}, method Accelerate() {...} >>]])); </pre>	<div style="display: flex; align-items: center; margin-bottom: 10px;"> <div style="font-size: 2em; margin-right: 10px;">⇒</div> <pre> function ABSCar(instance) { return [@instance : instance, method Move(...) {@instance.Move(...)}; method Accelerate (...) { @instance.Accelerate(...); }, method Break {...} ← refined method]; } </pre> </div> <div style="display: flex; align-items: center;"> <div style="font-size: 2em; margin-right: 10px;">⇒</div> <pre> function TurboCar(instance) { return [@instance : instance, method Break(...) { @instance.Break(...); }, method Move {...}, ← refined method method Accelerate {...} ← refined method]; } </pre> </div>
--	--

7.1.2 Adapter

The *Adapter* pattern allows converting the interface of a class into another interface that clients expect. This enables classes to work together that couldn't otherwise because of incompatible interfaces. The implementation of a generator for the Adapter pattern was already discussed in the case study of section 4.5.2. There we covered two implementation options, through subclassing and through delegation, however both in the context of an untyped language. Here we also provide a typed implementation

using the hypothetical meta-C++. Notice that in the context of a typed language, the AdapterGenerator metafunction (implemented as a functor class) requires knowledge about both the adaptee class and the target interface.

```
&class AdapterGenerator { ←pattern generator implemented as a functor class

public:
    typedef std::map<std::string, std::string> AdapterMap;

private:
    std::string      name;
    ClassAST*        target;
    ClassAST*        adaptee;
    AdapterMap        adaptedMethods;

public:
    AST* operator() () {
        AST* methods = (AST*) 0; ←will hold the adapter class method implementations
        const MethodList& targetMethods = target->GetMethods();
        for (MethodList::const_iterator i = targetMethods.begin();
            i != targetMethods.end(); ++i ←iterate over target class methods
        ) {
            assert((*i)->IsVirtual()); ←adaptation will work only on virtual functions
            const std::string name = (*i)->GetName();
            AdapterMap::const_iterator iter = adaptedMethods.find(name);
            const std::string originalName = iter == adaptedMethods.end() ?
                name : iter->second;
            assert(adaptee->GetMethod(originalName)->Matches(*i)); ←make sure
                the specified adaptee method matches the target method (same args & return type)

            Formals* formals = (*i)->GetFormals();
            AST* actuals = (AST*) 0; ←will hold the arguments for the adapted method call
            for (Formals::const_iterator j = formals->begin();
                j != formals->end(); ++j
            )
                actuals = <<~actuals, ~((*j)->GetName())>>; ←accumulate argument
            Type* returnType = (*i)->GetReturnType();
            AST* body = <<~(adaptee->GetName())::~originalName(~actuals)>>;
            ...the adapted method body consists only of the invocation of the original method...
            if (not returnType->IsVoid())
                body = <<return ~body;>>; ← handle non-void methods to propagate the result
            AST* method = <<~returnType ~name (~formals) { ~body; }>>;
            methods = <<~methods, ~method>>; ← create method and merge with existing
        }
        return <<
            class ~name: public ~(target->GetName()), ←inherit from target class
                private ~(adaptee->GetName()) { ←inherit from adaptee class
            public:
                ~methods; ←insert adapted methods
            };
        >>;
    }

    AdapterGenerator(const std::string& name, ClassAST* target,
        ClassAST* adaptee, const AdapterMap& adaptedMethods): name(name),
        target(target), adaptee(adaptee), adaptedMethods(adaptedMethods) {}
};
```

```

class OriginalWindow {                                     ←adaptee class
public:
    void Draw          (DC& dc)    {...}
    void SetWholeScreen (void)      {...}
    void Iconify       (void)      {...}
};

class TargetWindow {                                       ←target interface
public:
    virtual void Draw      (DC& dc);
    virtual void Maximize  (void);
    virtual void Minimize  (void);
};

&AdapterManager::AdapterManager adaptedMethods;
&adaptedMethods["Maximize"] = "SetWholeScreen";
&adaptedMethods["Minimize"] = "Iconify";
! (AdapterManager("Adapter", GetClass("TargetWindow"),
    GetClass("OriginalWindow"), adaptedMethods)());
class Adapter : public TargetWindow, private OriginalWindow {
public:
    void Draw      (DC& dc) { OriginalWindow::Draw(dc); }
    void Maximize  (void) { OriginalWindow::SetWholeScreen(); }
    void Minimize  (void) { OriginalWindow::Iconify(); }
};

```

7.1.3 Flyweight

The *Flyweight* pattern promotes the use of sharing to support a large number of similar objects efficiently. When an application requires a large number of objects that all involve similar state, repeating such state across every object could result in memory or efficiency problems. Instead, we can group the common state in a single immutable object called *flyweight* and share it across all target object instances that require it. This way, each object holds only its own mutable state along with a reference to the flyweight object thus significantly reducing the required memory.

A pattern generator for the flyweight pattern involves only specifying the contents of the flyweight object, i.e. the shared state. With this information, we can use a meta-function to traverse the involved class declarations and modify them to use the flyweight object instead of repeated state. This functionality is shown in the example below, again implemented in the hypothetical meta-C++.

```

&class FlyweightGenerator {                               ←pattern generator implemented as a functor class
private:
    static void ReplaceInAST(AST* ast, AST* original, AST* target);
    ← used to delegate all occurrences of a member to the flyweight object
public:
    typedef std::set<std::string> StringSet;

```

```

static AST* Apply (ClassAST* target, StringSet flyweightMembers) {
    AST* members = (AST*) 0;
    for (StringSet::const_iterator i = flyweightMembers.begin();
        i != flyweightMembers.end(); ++i) ←iterate over flyweight members
    ) {
        MemberAST* member = target->GetMember(*i);
        members = <<~members; ~member;>>
        ConstructorList& constructors = target->GetConstructors();
        for (ConstructorList::iterator j = constructors.begin();
            j != constructors.end(); ++j) ←iterate over target class constructors
            (*j)->RemoveFromInitializationList(*i);
        MethodList& methods = target->GetMethods();
        for (MethodList::iterator j = methods.begin();
            j != methods.end(); ++j) ←iterate over target class methods
            ReplaceInAST((*j)->GetBody(), ← delegate all occurrences of the member
                <<~(*i)>>, <<flyweight->~(*i)>>); ← to the flyweight object
            target->RemoveMember(*i);
    }
    target->AddInnerClass(<< ← create the AST of the Flyweight class
        struct Flyweight {
            ~members; ← members moved here from the target class
            static map<string, Flyweight*> pool; ← pool for flyweight instances
            static Flyweight* Get(const std::string& type);
        };>>);
    target->AddMember(<<Flyweight* flyweight;>>); ← insert flyweight reference
    const std::string name = target->GetName();
    return <<map<string, ~name::Flyweight*> ~name::Flyweight::pool;>>;
} ← all class functionality is inserted directly in the target class; return only the static map
instantiation that should be inserted outside the target class body (within a source file)
};

```

```

class Soldier {
private:
    State    state;
    Pos      pos;
    Task     task;
    Behavior behavior;
    Graphics graphics;
    AI       ai;
public:
    Behavior GetBehavior()
        { return behavior; }
    ...other public Soldier methods...
    Soldier(State s, Pos p, Task t):
        state(s), position(p), task(t),
        behavior(DEFAULT_BEHAVIOR),
        graphics(DEFAULT_GRAPHICS),
        ai(DEFAULT_AI) {}
};
std::set<std::string> members;
members.insert("behavior");
members.insert("graphics");
members.insert("ai");
! (FlyweightGenerator::Apply(
    GetClass("Soldier"),
    members
));

```

```

class Soldier {
private:
    State    state;
    Pos      pos;
    Task     task;
    struct Flyweight {
        Behavior behavior;
        Graphics graphics;
        AI       ai;
        static map<string, Flyweight*> pool;
        static Flyweight* Get(string type);
    };
    Flyweight* flyweight;
public:
    Behavior GetBehavior()
        { return flyweight->behavior; }
    ...other public Soldier methods...
    Soldier(State s, Pos p, Task t):
        state(s), position(p), task(t),
        flyweight(Flyweight::Get(DEFAULT))
        {}
};
map<string, Soldier::Flyweight*>
Soldier::Flyweight::pool;

```

7.1.4 Undo-Redo

Supporting undo/redo functionality is a common requirement for various applications, especially for interactive ones, with text editors and drawing applications being exemplifying examples. Such functionality can be achieved using the *Command* pattern as follows. The execute operation of a command can store state for reversing its effects, while the command interface offers an undo operation to reverse the effects of a previous execute call. Executed commands are stored in a history list that can be traversed backwards and forwards calling undo and execute (i.e. redo) respectively to support an unlimited number of undo and redo operations.

To generate a complete pattern implementation, apart from the Command interface and the command history list we also need the list of commands and the functionality they involve. Such information depends on the specific application context and has to be explicitly provided as deployment parameters to the pattern generator. In particular, when specifying the list of commands for a specific class that we want to extend with undo/redo functionality, we need to supply the data required by each command as well as the class methods that offer the matching functionality. Additionally, we should supply the logic for performing the undo operation. For instance, in a text editor containing methods `InsertText` and `DeleteText`, we may introduce an *Insert* command specifying the position where the insertion will take place as well as the text to be inserted, and a *Delete* command specifying the starting position for the deletion and the number of characters to delete. Additionally, for the *Insert* command the execute operation should be associated with `InsertText` and its undo operation with `DeleteText`. Similarly, for the *Delete* command the execute operation should be associated with `DeleteText` and its undo operation with `InsertText`. Finally, the original class methods should be implemented through the available commands so as to offer the undo/redo functionality. With such information specified as deployment parameters, we can implement a pattern generator meta-function able to support various undo/redo application instances as follows. This is illustrated in the following example, where the pattern is delivered as a subclass that offers the required undo/redo functionality and overrides the undoable methods to issue matching commands.

```

&function CreateCommand(name, data, execute, undo) {
    local args = nil, local formals = nil, local members = nil;
    foreach(local item, data) {
        formals = <<~formals, ~expr>>;          ←merge command data for formal list
        args = <<~args, self.~item>>; ←merge command data for the execute invocation
        members = <<~members, {~item:~item}>>; ←merge command data for members
    }
    return <<method ~name(object, ~formals) { ← command constructor function
        return [
            @object : object,
            ~members,
            method execute() {
                ~(execute.storeUndoData); ←code for storing the undo data
                self.object..~(execute.method) (~args);
            }, ←perform the execute action call locally on the target object
            method undo() ←perform the undo action call locally on the target object
            { self.object..~(undo.method) (~(undo.args)); }
        ];
    }>>;
}

&function OverrideFunc(name, command) {
    return <<method ~name(...) { ←generate the overridden method
        local base = std::getbase(self, 0); ←get base class object
        @newcommand(@commands.~command(base, ...));
    }>>; ←create the matching command targeting the base class object
}

&function UndoRedo(name, commandSpecs, funcMappings){ ←pattern generator
    local commands = nil; ←will hold the command class constructor functions
    foreach(local spec, commandSpecs) { ←create command classes based on specs
        cmd=CreateCommand(spec.name, spec.data, spec.execute, spec.undo);
        commands = <<~commands, ~cmd>>; ←merge commands
    }

    local overrides = nil; ←will hold the overridden version of the undoable functions
    foreach(local map, funcMappings) ←for all specified funcs create and merge overrides
        overrides=<<~overrides, ~(OverrideFunc(map.method, map.command))>>;

    return <<
        function ~name(baseObject){←generate a derived class with undo/redo functionality
            local objectWithUndoRedo = [
                @commands: [~commands], ←holder for the command class constructors
                @undoStack : list_new(), ←use 2 stacks to implement the command history
                @redoStack : list_new(),
                method undo {
                    if (@undoStack.total() > 0) { ←if there are actions to be undone
                        cmd = @undoStack.pop_front(); ←remove command from undo stack
                        @redoStack.push_front(cmd); ←insert the command in the redo stack
                        cmd.undo(); ←undo the effects of the command
                    }
                },
                method redo {
                    if (@redoStack.total() > 0) { ←if there are actions to be redone
                        cmd = @redoStack.pop_front(); ←remove command from redo stack
                        @undoStack.push_front(cmd); ←insert the command in the undo stack
                        cmd.execute(); ←execute the command again
                    }
                },
            ],
        }
    >>
}

```

```

        method newcommand(cmd) {      ← overridden methods just issue a new command
            @undoStack.push_front(cmd);    ← insert the command in the undo stack
            @redoStack.clear();    ←no undone actions can be redone after new actions
            cmd.execute(); ←execute command to perform the original method functionality
        },
        ~overrides                    ←insert the overridden methods in the subclass
    ];
    inherit(objectWithUndoRedo, baseObject);
    return objectWithUndoRedo;
}
>>;
}
&commandSpecs = [
    [
        @name : "Insert",                ←command class name
        @data : list_new("pos", "text"), ←command class data
        @execute : [/*@storeUndoData : nil,*/ ←code for storing undo data(here none)
            @method: "AddText"], ←original method for execute action(args implied from data)
        @undo : [@method : "DeleteText", ←original method for undo action
            @args : <<@pos, strlen(@text)>>] ←arguments for undo action
    ],
    [
        @name : "Delete",
        @data : list_new("pos", "len"),
        @execute : [@storeUndoData : <<@text = @object.text;>>,
            @method : "DeleteText"],
        @undo : [@method : "AddText",
            @args : <<@pos, strslice(@text, @pos, @pos + @len - 1)>>]
    ],
    [
        @name : "Set",
        @data : list_new("text"),
        @execute : [@storeUndoData : <<@old_text = @object.text;>>,
            @method: "SetText"],
        @undo : [@method : "SetText", @args : <<@old_text>>]
    ]
];
&funcMappings = [                ←map original object methods to matching commands
    [ @method : "AddText", @command : "Insert"],
    [ @method : "DeleteText", @command : "Delete"],
    [ @method : "SetText", @command : "Set"]
];
function Editor() {                ←original class with no undo/redo functionality
    return [
        @text : "", @caret : 0,
        method Display (msg) {...},
        method SetText (text) {...},
        method AddText (pos, text) {...},
        method DeleteText(pos, len) {...}
    ];
}
! (UndoRedo(<<EditorWithUndoRedo>>, commandSpecs, funcMappings));
←generates the derived class EditorWithUndoRedo that offers the required undo/redo functionality
editor = EditorWithUndoRedo(Editor());
editor.AddText(0, "Hello world!");
editor.undo(); editor.redo();

```

7.2 Exception Handling Templates

Exception handling [Goodenough] is a key mechanism supporting structured error recovery within a software system. It allows effectively decoupling normal code from error handling code through distinctive *try-catch* control blocks. Once exceptions are raised with a *throw* statement, the program control is transferred to the closest handler matching the raised exception. The same exception raised by different contexts may naturally be caught by different handlers, something dependent on the calling context which led to the statement raising the exception.

Ideally, exception handling in a language should facilitate the construction of handling code in a way that is modular and easy to reuse. But still the challenge of implementing modular, generic and directly reusable exception handler code remains an open issue for software developers. The main problem is that in real-life software systems the normal code and the handling code are frequently tightly coupled. An important factor in this context is that the exception handling logic is encompassed within syntactically distinct blocks, meaning the chances for applying language reuse approaches such as inheritance, abstraction, polymorphism and genericity, become de facto restricted. Finally, although there are well known good and bad practices regarding exception handling [Doshi][McCune], none of the established exceptions patterns can be directly reused in an implementation form across application contexts. In this context, we thought that once we manage to parameterize the syntactic structures, essentially treating syntax, and in effect source code, as a first-class value, we may succeed in realizing exception patterns in a directly implementable and deployable form.

This line of thinking has driven us to consider staged metaprogramming. Using metaprogramming, it is possible to express error handling patterns as parameterized source fragments in the form of *AST templates* with empty placeholders ready to host client-supplied code. Each pattern is implemented by a respective metafunction that can be invoked during compilation at the appropriate source context with the desirable parameters in order to generate a concrete instantiation of the exception handling pattern. Such patterns may in general encompass complex exception handling structures, combining arbitrary statements with nested and repeated try-catch blocks (Figure 7.1).

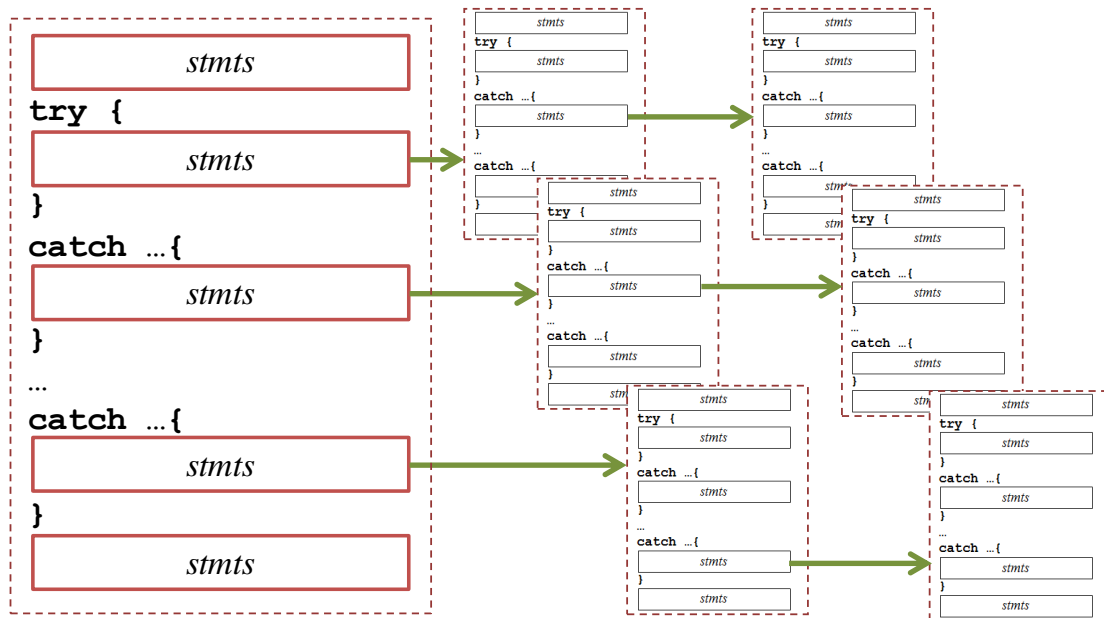


Figure 7.1 – The recursively-repeated structure of exception handling patterns generated via metaprogramming; *stmts* may contain exception handling code following the main pattern.

Nevertheless, the client code need only be aware of the pattern behavior, its corresponding meta-function and how to apply it. This way, not only programmers are relieved from underlying, sometimes transient, implementation details, but the exception handling patterns can be standardized and be directly reused. In this sense, it is possible to produce libraries of parameterized exception handling patterns, thus allowing configuring pattern deployment at generation-time based on the particular application requirements. This property is illustrated in Figure 7.2. We support our

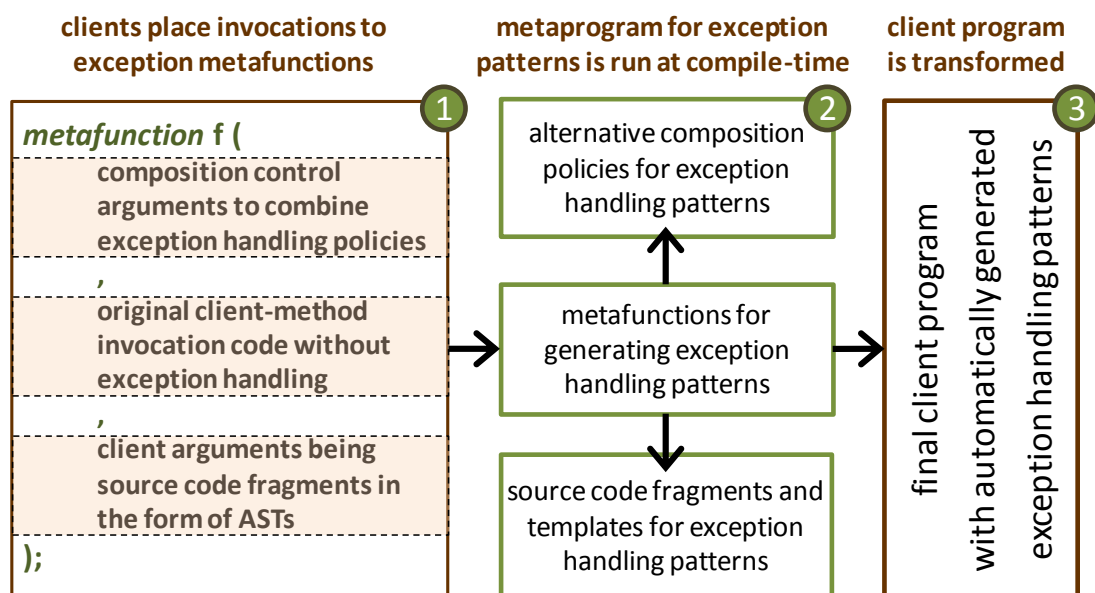


Figure 7.2 – Deploying library metaprograms to generate exception handling patterns.

claim by examining key exception handling scenarios and by providing all respective meta-implementations to eventually turn them into reusable pattern libraries.

Since Delta is an untyped language, the implementations discussed in most scenarios correspond to untyped exception handling. Nevertheless, our approach has nothing to do with type checking or type systems, so it can be applied to any language, either typed or untyped, as long as it offers the required metaprogramming features. In particular, some of the presented scenarios involve typical staged computations so they could be directly adopted in any of the multi-stage (or even two-stage) languages discussed earlier (e.g. MetaML, Nemerle, MetaLua, Converge, Groovy, etc.). However, there are other scenarios involving state sharing and control flow in stage code (like the one discussed in the case study of section 4.5.1), making them more suitable for the integrated metaprogramming model.

7.2.1 Exception Handling Scenarios

We discuss a wide range of exception handling scenarios including resource failures, high level architectural exceptions, and multiple repeating catch blocks. DbvC contractual exceptions also fall in this category, but were already discussed in the case study of section 4.5.3, so they are not repeated here. Each of the discussed scenarios as well as other exception handling patterns discussed later can be modularly organized by collecting all relevant meta-functions in library modules. Then any client module can simply deploy these libraries during its compilation; as previously discussed, apart from the normal (i.e. runtime) dependencies of a program, it may also have meta-code dependencies, i.e. dependencies for some of its stages.

Resource failures. A typical error handling category relates to exceptions being raised due to resource failures. This includes scenarios where the system runs out of memory, a network connection fails, or a local database does not respond properly. Through metaprogramming we can reduce the boilerplate code required to test various resource failures, by automatic generation at the desirable client context. An implementation and its deployment are provided below.

```
&function Resources (alloc_stmts, handler_stmts)
  { return <<try { ~alloc_stmts; } catch e { ~handler_stmts; }>>; }
&function InitialMemoryAllocation (alloc)
  { return Resources(alloc, <<print("No memory!"); exit();>>; }
&function NormalMemoryAllocation (alloc)
  { return Resources(alloc, <<Collector(); ~alloc;>>; }
```

```

!(InitialMemoryAllocation(<< x = malloc(10) >>));
!(NormalMemoryAllocation(<< y = malloc(20) >>));
try { x = malloc(10); } catch e { print("No memory!"); exit(); }
try { y = malloc(20); } catch e { Collector(); y = malloc(20); }

```

High level architectural exceptions. When implementing the interaction amongst high-level architectural components, the exceptions that can be raised are usually formalized and relate to predefined conditions that may fail during runtime. Along these lines, a component is aware of the exceptions that may be raised by a certain invocation and its reaction to them is typically predetermined: it either knows how to handle errors, in which case it deals with them directly, or it does not and just filters and propagates the exceptions to the calling component. These high-level exception interactions can be turned into metacode, inserted at the appropriate sites of component implementations. This allows standardizing exception interactions as component meta-data and can lead to cleaner code that is easier to understand and maintain. Such functionality can be achieved with the exception pattern shown below.

```

&function ArchitecturalException (exception) {
  return [ ← returns a generator instance with meta methods for architectural exceptions
    method Raise { return << throw ~exception; >>; },
    method Ensure(condition) ← if the condition fails will raise an exception
      { return << if (not (~condition)) ~(self.Raise()); >>; },
    method Filter(invocation_stmts, filter_stmts) {
      return << ← upon an exception execute the filtering statements and rethrow
        try { ~invocation_stmts; }
        catch e { assert e==~exception; ~filter_stmts; throw e; }
      >>;
    },
    method Handle(invocation_stmts, handler_stmts) {
      return << ← handle an exception executing the given handler statements
        try { ~invocation_stmts; }
        catch e { assert e == ~exception; ~handler_stmts; }
      >>;
    }
  ];
}

```

An invocation example along with the source code it generates is provided below:

```

&bank = ArchitecturalException("NegativeAmount");
!(bank.Ensure(<<acc.amount >= 0>>));
!(bank.Handle(<<acc.Withdraw(100)>>, <<acc.CancelTransaction()>>));
if (not (acc.amount >= 0)) throw "NegativeAmount";
try { acc.Withdraw(100); }
catch e { assert e == "NegativeAmount"; acc.CancelTransaction(); }

```

Multiple repeating catch blocks. A common scenario encountered in exception handling relates to small source fragments, even single statements, which may raise

multiple distinct exceptions. Such code fragments may be found in various independent locations of the program but the error handling strategy is usually similar. A typical example of this scenario is the use of sockets, where various things may go wrong (e.g. *IOException*, *TimeoutException*, *SecurityException*, etc.), but each of them is usually handled in a custom manner. For instance, upon a *TimeoutException* the program will typically wait and retry the operation after some time, upon an *IOException* it will try to reestablish the IO stream, upon a *SecurityException* it may try to elevate security privileges or notify the user about insufficient permissions, and so on. Such cases require introducing comprehensive cascading catch blocks that cannot be abstracted via polymorphism or genericity in any manner. But it is possible to introduce metafunctions capturing the cascaded exception handling logic and insert it at the appropriate client sites thus accomplishing the desirable exception handling behavior. This is demonstrated by the following code:

```
&function Cascading (invocation_stmts, alt_handlers) {
  local ast = nil;
  foreach (typeof exception : handlerfor exception, alt_handlers)
    ast = <<<create entry to map exception type to handler and merge with previous entries
      ~ast, { ~type : function{ ~handler; } }
    >>>;
  return <<
    try { ~invocation_stmts; }
    catch e {
      local D = [~ast];
      local f = D[e.type];
      if (f) f(); else throw e;
    }
  >>>;
}

&FILE_IO_Handlers = [
  { "EOFException" : << reader.close(); >> },
  { "FileNotFoundException" : << print("no file"); >> },
  { "UnknownEncodingException" : << load_encodings(); >> }
];

!(Cascading(
  << reader = FileReader("foo.abc"); reader.read(); >>,
  FILE_IO_Handlers
));
try { reader = FileReader("foo.abc"); reader.read(); }
catch e {
  local D = [
    { "EOFException" : function { reader.close(); } },
    { "FileNotFoundException" : function { print("no file"); } },
    { "UnknownEncodingException" : function { load_encodings(); } }
  ];
  local f = D[e.type];
  if (f) f(); else throw e;
}
```

↓

Our examples are untyped so the cascading catch blocks are modeled through a single catch block, using a dispatcher to choose a handler via the runtime exception type tag. In a typed language, the generated code would consist of successive catch blocks for typed exceptions, each with the respective handler invocation. For example, the following code implements the above scenario in a hypothetical meta-Java language adopting our staging tags and programming model.

```
&class Cascading {
  public static AST generate(AST stmts, HashMap<String,AST> handlers) {
    TryCatchAST ast = new TryCatchAST(); ← class to create AST for a try-catch block
    ast.setTryBlock(<< try { ~stmts; } >>); ←try blocks
    for (Map.Entry<String, AST> entry : handlers.entrySet()) {
      String type = entry.getKey();
      AST handler = entry.getValue();
      ast.addCatchBlock(<< catch (~type e) {~handler;} >>); ←catch blocks
    }
    return ast;
  }
}

&HashMap<String, AST> SOCKET_Handlers = new HashMap<String, AST>();
&SOCKET_Handlers.put("UnknownHostException", <<...>>);
&SOCKET_Handlers.put("ConnectException", <<...>>);
&SOCKET_Handlers.put("NoRouteToHostException", <<...>>);
&SOCKET_Handlers.put("IOException", <<...>>);

Socket s = null;
! (Cascading.generate(
  <<s = new Socket("host", 80);>>,
  SOCKET_Handlers
)); ← produce the cascading handling logic
```

```
Socket s = null;
try { s = new Socket("host", 80); }
catch (UnknownHostException e) {...}
catch (ConnectException e) {...}
catch (NoRouteToHostException e) {...}
catch (IOException e) {...}
```

7.2.2 Exception Policies

We recall the case study of section 4.5.1 illustrating the adoption of metaprogramming to select different exception handling policies for different parts of a system. The Logging and Retry meta-functions shown there are just two of the policy examples that we have implemented. Below we present a more elaborate set of exception handling policies incorporating strategies discussed in [WirfsBrock], and then implement each policy using a corresponding meta-function.

- *None* – Do not handle exceptions
- *Inaction* – Ignore any raised exceptions
- *Logging* – Log any raised exceptions
- *Retry* – Repeatedly attempt an operation that raised an exception
- *Rollback* – Try to proceed, but on failure undo the effects of the failed action

- *Cleanup Rethrow* – Perform any cleanup actions and propagate the exception
- *Higher Level Exception* – Raise a higher level exception
- *Guarded Suspension* – Suspend execution until a condition is met and retry

```

&function None (stmts) { return stmts; }      ← no handling, just return stmts
&function Inaction (stmts) { return << try { ~stmts; } catch e {} >>; }
&function Logging (stmts)
{ return << try { ~stmts; } catch e { log(e); } >>; }
&function CreateRetry (data) {                ← constructor for a custom retry policy
return function (stmts) {                    ← return a function implementing the code pattern
return <<                                    ← the returned function returns an AST
for (local i = 0; i < ~(data.attempts); ++i)
try { ~stmts; break; }                      ← try & break loop when successful
catch e { Sleep(~(data.delay)); }           ← catch & wait before retrying
if (i == ~(data.attempts))                  ← maximum attempts were tried?
{ ~(data.failure_stmts); }                  ← then give-up & invoke failure code
>>;
};
}
&function CreateRollback (rollback_stmts) {
return function (stmts) {
return <<
try { ~stmts; }                             ← try the code and on error execute the rollback stmts
catch e { ~rollback_stmts; }
>>;
};
}
&function CreateCleanupRethrow (cleanup_stmts) {
return function (stmts) {
return <<
try { ~stmts; }                             ← try the code and on error cleanup and rethrow
catch e { ~cleanup_stmts; throw e; }
>>;
};
}
&function CreateHigherLevelException (exception) {
return function (stmts) {
return <<
try { ~stmts; }                             ← try the code and on error throw a higher-level exception
catch e { throw [ ~exception, @source : e ]; }
>>;
};
}
&function CreateGuardedSuspension(condition) {
return function (stmts) {
return <<
while (true)
try { ~stmts; break; }                      ← try the given code and break on success
catch e { while (not ~condition); }         ← else wait condition to hold
>>;
};
}

```

The policy implementations are straightforward, typically placing the given invocation code (stmts) in a try block and the handler logic in a catch block.

The deployment is similar to that shown in section 4.5.1, with each policy details specified only once at their registration and then reused across all generated *catch* blocks. The latter enables exception handling policies to become standardized library components that can be reused based on the application requirements.

An alternative approach for parameterized exception handling policies [Newton] involves a library filtering already caught exceptions. This means that boilerplate code is repeated per handler, but more importantly it cannot support scenarios with more elaborate exception handling logic (for example *Retry*, *Guarded Suspension*, etc.).

7.2.3 Process Modeling Patterns

Exception handling is not limited to the scope of specific functions or software modules, but also applies in the more general context of a process model. In both cases it is important to specify the normal execution path as well as the possible exceptional behaviors along with the tasks required to handle them properly. In this sense, exception handling patterns observed in the one world can also be beneficial to the other. To this end, we adopt the *trying other alternatives* and *inserting behavior* process modeling patterns shown in [Lerner] and implement them as meta-functions.

Trying other alternatives. It is possible for a single task to be accomplished in multiple ways, possibly involving different components and relying on different conditions. Instead of explicitly using such information in the code structure, it is preferable to abstract the functionality in distinct operations, where all of them achieve the same task, and if one fails another alternative may be tried in its place. Below we provide an exception handling pattern implementation for this scenario. The *alternatives* argument contains a list with the alternative code fragments for the given task, while the *ex* argument is the exception that signals failure of a task so as to try an alternative.

```
&function TryAlternatives (alternatives, ex) {
  local ast = << local success = false; >>; ← guard for successful alternative
  foreach (local alt, alternatives)
    ast = << ~ast;                               ← merge with previously produced AST
    if (not success)                               ← skip rest of alternatives if task has succeeded
      try { ~alt; success = true; } ← if no exception we succeeded
      catch e { if (e != ~ex) throw e; } ← throw all other exceptions
  >>;
  return << ~ast; if (not success) throw ~ex; >>; ← throw if all fail
}
```

An example of invoking the *TryAlternatives* meta-function at compile-time and the respective source code it introduces in its place is provided below:

```
&alternatives = list_new(<<hotel1.Book()>>, <<hotel2.Book()>>);
&Order(alternatives);    ← optionally apply client-specific ordering
!(TryAlternatives(alternatives, "FullyBookedException")); ↓
local success = false;
if (not success)
    try { hotel1.Book(); success = true; }
    catch e { if (e != "FullyBookedException") throw e; }
if (not success)
    try { hotel2.Book(); success = true; }
    catch e { if (e != "FullyBookedException") throw e; }
if (not success) throw "FullyBookedException";
```

Inserting behavior. When errors occur during the execution of a series of tasks, they may not be fatal, but may instead require specific actions to be performed to fix the problems that caused them. This inserted behavior may have to be executed directly after the occurrence of an error before any later tasks are executed (*immediate fixing*), or it may be possible to just note the error and handle it accordingly after all tasks are completed (*deferred fixing*). Below we provide a pattern implementation supporting both scenarios. The *tasks* argument contains all relevant information (normal code, exceptions that they may raise and handler code) about the tasks to be executed, while the *immediate* argument specifies when to apply the error handling code.

```
&function InsertBehavior (tasks, immediate) {
    local ast = nil, local err = nil, local i = 0;
    foreach (local task, tasks) {
        ast = << ~ast;                                ← merge with previously produced AST if any
        try { ~(task.stmts); }                          ← insert the task-related stmts
        catch e {
            if (e != ~(task.except)) throw e;    ← throw all other exceptions
            ~(immediate ? task.handler : << errors[~i] = true >>);
        }      ← if immediate-fixing insert directly the task handler else record the error
    >>;
    if (not immediate)    ← if deferred fixing insert code to handle recorded errors
        err = << ~err; if(errors[~i]) { ~(task.handler); } >>;
    ++i;
}
return (immediate ? ast : << local errors=[]; ~ast; ~err >>);
}
```

```
&tasks = list_new(
    [ @stmts : << LoadConfig()>>,
      @except : "NotFound",
      @handler: <<LoadDefaultConfig()>> ],
    [ @stmts : <<SendData()>>,
      @except : "ConnectionError",
      @handler: <<RepairConnection()>> ]
);
!(InsertBehavior(tasks, true));
```

```
try { LoadConfig(); }
catch e {
    if(e != "NotFound") throw e;
    LoadDefaultConfig();
}
try { SendData(); }
catch e {
    if(e != "ConnectionError")
        throw e;
    RepairConn();
}
```

7.2.4 Pattern Combinations

The previously discussed patterns target different implementation layers ranging from low level operations on sockets to high level component interactions towards a common task. All these patterns are orthogonal and can be combined with each other to form more elaborate and custom exception handling styles (Figure 7.3).

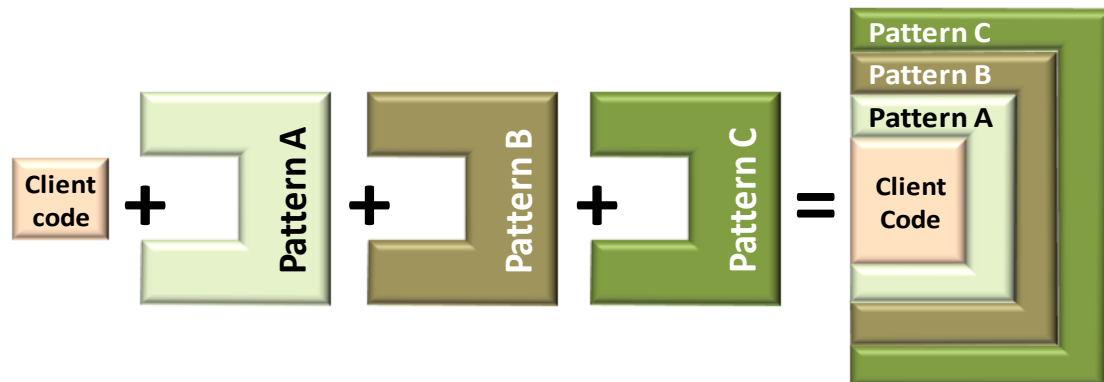


Figure 7.3 – Modular composition of exception-handling patterns as decorator stacks.

It is important to note that using this approach the pattern combination maintains a linear code complexity level. Even though the generated code may contain multiple levels of nested and/or cascading catch blocks, the original code involves mainly independent pattern implementations, typically provided as a library, and the pattern combination that essentially behaves as a code decoration process. For example, consider the task of booking a hotel. Various alternative hotel objects may be available for booking, each object involving Design by Contract tests, while the system may adapt a retry policy to handle any raised exceptions. Clearly, addressing all these requirements by manually inserting exception handling code would result into error-prone code that will be hard to read, understand and maintain. However, using metaprogramming we can combine applications of the *Retry Policy*, *Try Alternatives* and *Design by Contract* patterns.

```
! (ExceptionPolicies.Get("RETRY") (
    TryAlternatives(
        list_new(
            DbyC().client(<<hotel1.Book()>>),
            DbyC().client(<<hotel2.Book()>>),
        ), "FullyBookedException"
    )
));
```

←apply the *Retry* pattern
 ←apply the *Try Alternatives* pattern
 ←apply *DbyC* pattern
 ←apply *DbyC* pattern

↓


```

for (local i = 0; i < 5; ++i)
  try {
    local success = false;
    if (not success)
      try {
        try { hotel1.Book(); }
        catch ContractException { log(ContractException); }
        success = true;
      } catch e { if (e != "FullyBookedException") throw e; }
    if (not success)
      try {
        try { hotel2.Book(); }
        catch ContractException { log(ContractException); }
        success = true;
      } catch e { if (e != "FullyBookedException") throw e; }
    if (not success) throw "FullyBookedException";
    break;
  } catch e { Sleep(1000); }
if (i == 5) { post("FAIL"); }

```

Annotations in the code block:

- ↙ (pointing to the outer try block): *← try-catch generated by Retry*
- ↙ (pointing to the inner try block): *← try-catch generated by TryAlternatives*
- ↙ (pointing to the inner try block): *← try-catch generated by DbyC*
- ↙ (pointing to the inner try block): *← try-catch generated by TryAlternatives*
- ↙ (pointing to the inner try block): *← try-catch generated by DbyC*

7.2.5 Comparison to AOP

Our proposition for exception handling templates targets the delivery of modular and reusable error handling code. As such, it is closely related to AOP and the work towards separating the exception handling logic from the application code and modularizing it into aspects.

Introductory texts [Kiczales97][Laddad03] describe exception handling as a potential target for applying AOP and there are refactoring catalogues [Cole][Laddad06] that include procedures for moving exception-handling code to aspects; however they do not assess the suitability or effectiveness of the approach. An initial study on this subject [Lippert] showed that aspects can decrease the number of LOC, but later more in-depth studies [Filho06a][Filho06b] showed that there are cases where aspects may bring more harm than good. In general, current AOP languages have some limitations when used for exception handling. Firstly, they cannot express certain exception handlers without leading to program anomalies [Filho07]. Secondly, they do not help much in making the interface between normal and error handling code explicit [Filho06b]. Additionally, turning context dependent handlers into aspects requires introducing artificial changes, thus causing software maintenance issues [Greenwood][Filho09]. Finally, mixing exception handling with AO programs can hinder program reliability as the exception flow is complicated, leading to several possible bugs [Coelho08A][Coelho08B].

Using metaprogramming to generate exception handling code structures can help overcoming limitations found in AO solutions. Firstly, the exception handling code is generated *in-place* meaning that any required context is directly available to it, avoiding the need for explicitly providing it as additional aspect parameters. Secondly, code generation allows creating any code structure including nested and cascading exception handlers without having to specify multiple advices. Additionally, combining multiple exception handling patterns is explicit and straightforward; there can be no conflicts from independently deployed aspects where no ordering is specified. Finally, metaprogramming allows parameterizing code structures and thus combining similar functionality, something not always possible through typical AO advice [Filho09]. Table 7.1 highlights the pros and cons of each approach with respect to exception handling.

Table 7.1 – Comparison of AOP and metaprogramming in the context of exception handling.

	Aspects	Metaprograms
Automation	✓ Pointcuts match multiple sites	✗ Pattern is repeated per site
Combination	✗ May impose explicit ordering	✓ Free user-defined ordering
Context	✗ Handlers depending on local context break encapsulation	✓ Handlers are generated in-place and always access local context
Reuse	✗ Similar code fragments must be repeated every time ✓ Reuse via aspect inheritance	✓ Similar code fragments can be composed and reused as ASTs ✓ Reuse via metafunctions
Expressiveness	✗ Bound by pattern matching	✓ Allows any handler scenario

7.3 Staged Model-Driven Generators

The general philosophy of Model-Driven Engineering (MDE) [Kent][Schmidt] rents its roots to Model-Driven-Architecture (MDA) [OMG10] of the Object Management Group, emphasizing rapid application development together with model-oriented reuse and evolution. The core idea is that it is possible to capitalize on platform-independent models, use them to automatically derive platform-specific models through transformation engines and ultimately utilize code generators to automatically produce the source code corresponding to the modeled entities. The generated source code can then be extended or linked with custom application code to deliver the final application (Figure 7.4).

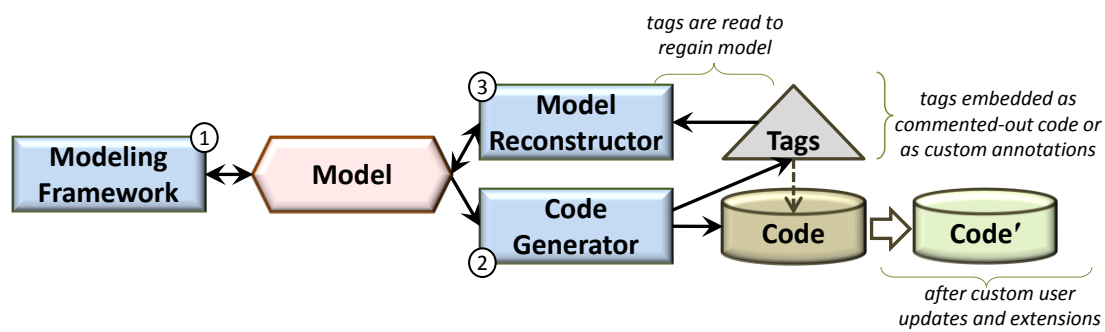


Figure 7.4 – Architecture of generative model-driven tools: (1) interactive model editing; (2) code generation from models; and (3) tags inserted in the generated source code to carry model information and enable model reconstruction.

Custom user updates or extensions may introduce two maintenance issues once code is freely edited: (i) if source tags are affected model reconstruction is broken; and (ii) code inserted without special tags is overwritten on regeneration. To address these issues we investigate an alternative path where the output of an MDE tool becomes available in the form of an AST and it is inserted along with normal application code *on-demand* and *in-place* through staged metaprogramming. This work has been conducted as a separate Master's Thesis exploring the adoption of metaprogramming in the field of MDE. Here we only give an overview of the main ideas involved and the proof-of-concept prototypes. For a more detailed discussion see [Valsamakis].

7.3.1 MDE Maintenance Issues

MDE tools cannot optimally address all required features of an application at the software engineering level. Thus, custom source code amendments and modifications

are always anticipated. Even if advanced methods are deployed to modularize and decouple the generated code from the rest of the application code, one can never exclude the possibility that interdependencies or custom updates may appear.

The typical lifecycle of the generated code is outlined under Figure 7.5. As shown, a dependency is introduced by having the application logic directly refer and deploy generated components (middle part). But for most languages this is overall insufficient for effectively linking application and generated code, practically requiring the generated code to be also manually modified. Typical updates relate to application functionality importing and invoking, application-specific event handling, linkage to third-party libraries that are not known to the model-driven tool, code improvement or refactoring. This situation very quickly results into many bidirectional dependencies (right part).

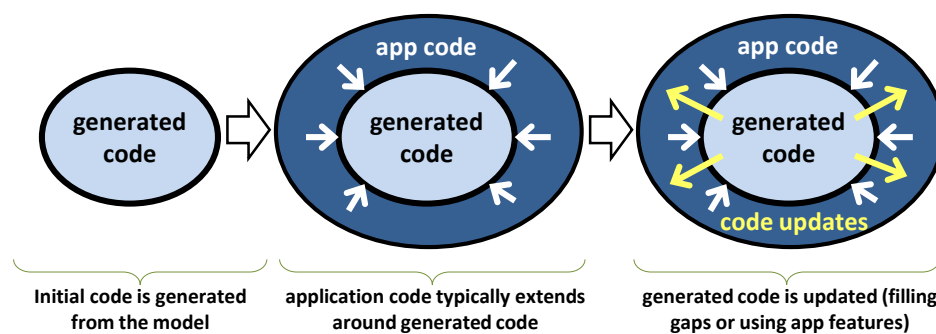


Figure 7.5 – Common growth of application code around the originally generated code; future custom extensions and updates eventually lead to bidirectional dependencies.

The latter maintenance issues are detailed in the typical generative model-driven process shown in Figure 7.6. Initially, if the code is not changed, source regeneration and model reconstruction are well-defined (left, steps 1-4). In other words, the MDE tool works perfectly for both steps of the processing loop. However, once the generated code is updated (left, step 5), two problems directly appear. Firstly, tag editing and misplacing may break model reconstruction (left, steps 6-7), while any code manually inserted outside the MDE tool causes a model-implementation conflict. Secondly, source regeneration overwrites all manually introduced updates (left, steps 8-9). For real-life applications of a considerable scale the latter may lead to the adoption of the MDE tool only for the first version, or worse, avoiding using an MDE tool at all.

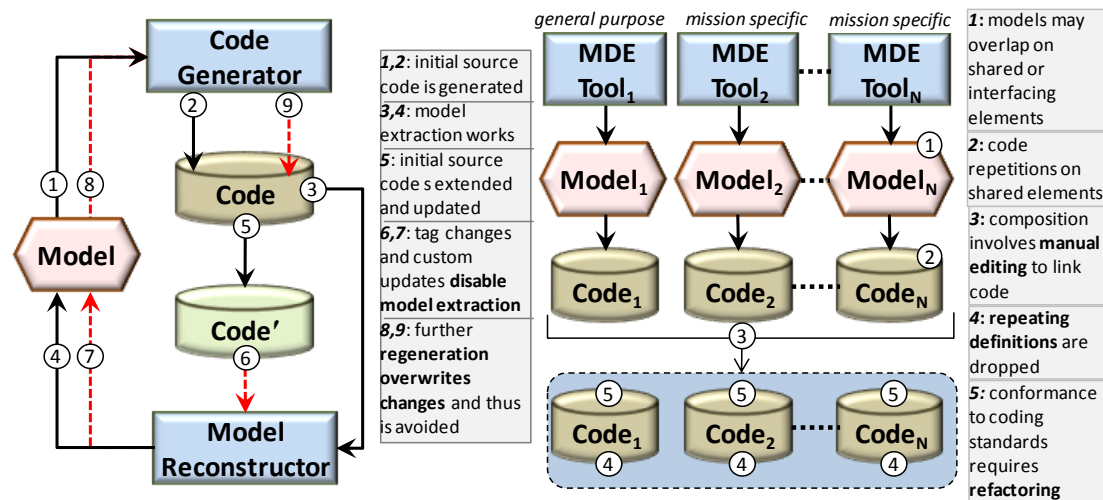


Figure 7.6 – The primary maintenance issues in the deployment of generative model-driven tools either individually (left) or collectively (right).

Maintenance issues also arise when trying to combine the outcome of multiple MDE tools. When using multiple tools, a single application element may end up being shared by different models. This means that when the code for each model is generated, there will be code repetitions for the shared elements (right, steps 1-2). In this case, the developer has to manually edit the generated sources to drop any repeated definitions and link the code properly (right, steps 3-4). Furthermore, the use of different MDE tools implies different code generators and thus different coding styles and methods present in the generated code. Having all generated sources conform to specific coding standards inevitably requires manual refactoring (right, step 5).

7.3.2 Improving the MDE Process

To address the inherent maintenance issues involved in the deployment of generative MDE tools we started thinking of an alternative path, in which the MDE tool output would somehow remain invariant, i.e. in a not-editable form, and the source code of the application could still grow and evolve in an unconstrained manner around it. This led us to the idea of bringing staging into the pipeline by enabling programmers algorithmically manipulate the generated code including: loading, processing and transforming. This approach, not only addresses the maintenance issues of traditional generators, but also sets code manipulation as a first-class concept in MDE and reveals the value of using a metaprogramming language in this context. In this direction, we discuss two deployment options: (i) *languages with explicit stages*

(including all multi- and two- stage languages discussed earlier), providing maximum compositional flexibility for source code manipulation and insertion; and (ii) languages utilizing their reflection API to support *implicit staging* (e.g. Java, C#, Python, Lua, Ruby, etc.), providing a less flexible but still powerful and feasible solution.

7.3.2.1 Deploying Staging

The refined model-driven process with staging is outlined under Figure 7.7. As shown, the first step concerns stage code evaluation that inserts the model code along with the normal program code, while the second one concerns normal program translation or evaluation. In particular, with staged model-driven generation the MDE process is improved as follows. Initially, the model-driven tools generate code in the form of language-specific ASTs. Apart from code, the ASTs can also incorporate any special code annotations, like those required by various Java frameworks. ASTs are essentially read-only data, meaning the result of the code generation remains unchanged and thus the code-to-model reconstruction path is unnecessary. Then, the generator macros are evaluated, reading and manipulating the ASTs as previously discussed, and finally inserting the desired source fragments where needed. This method fully supports the manipulation of multiple ASTs regardless of their originating tool. Essentially, this allows for combined deployment of different MDE

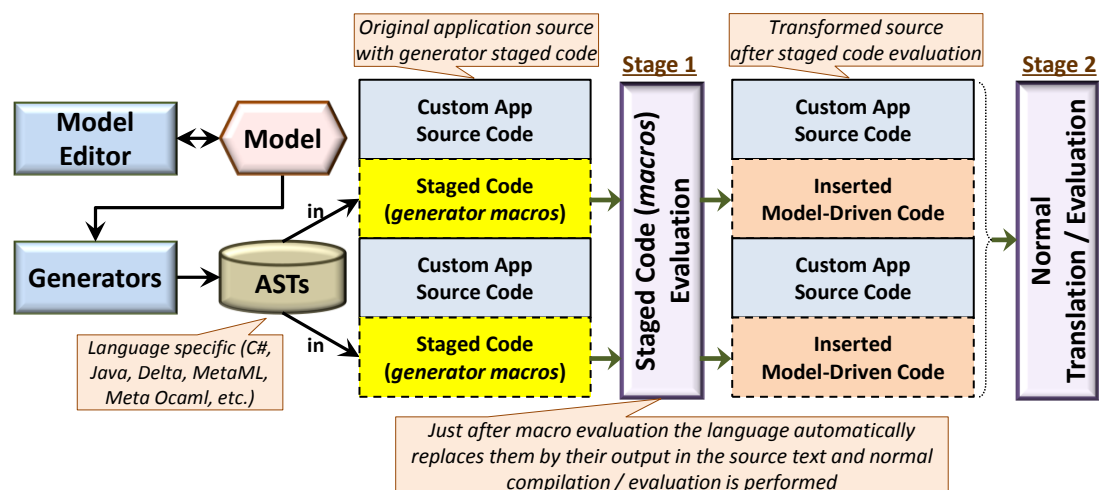


Figure 7.7 – The refined model-driven process with an inverted responsibility through staging: programmers deploy generator macros to insert generated code on-demand and in-place without affecting the originally produced ASTs by the MDE tools. The second stage applies translation on compile-time staging, or evaluation (translation and execution) on runtime staging.

tool outputs. Additionally, the generator macros may contain any application-specific composition or editing logic. This means that it is possible to perform any code transformation on a source fragment before inserting it in the final source. Finally, after the staged evaluation has produced the final source, the process continues with the normal translation (compile-time staging) or evaluation (runtime-staging).

7.3.2.2 Deploying Reflection

Many popular languages do not support staging; nevertheless, one may deploy the reflection mechanism of languages like C# or Java to practice a similar source code management and generation pipeline as the one discussed in the previous section. This option is detailed under Figure 7.8, showing that the language compiler and the dynamic class loading and method invocation facilities (i.e. reflection API) are directly deployed. The entire process starting the conversion from ASTs to intermediate representations (very flexible, suggested), or alternatively to source text (more rigid, not suggested), should be explicitly implemented as it is not automated by the languages. However, it is cached, meaning it is not repeated during execution, but applied once per AST version.

The oval of Figure 7.8 labeled as *composition parameters* represents the need for performing custom mixing between the automatically generated source code and the manually inserted code, something that is apparent in the presence of *Composer* as an integral part of the application. This is similar to AST composition alternatives, although at the intermediate representation level, and is very critical to ensure that maximum code mixing freedom is provided to developers.

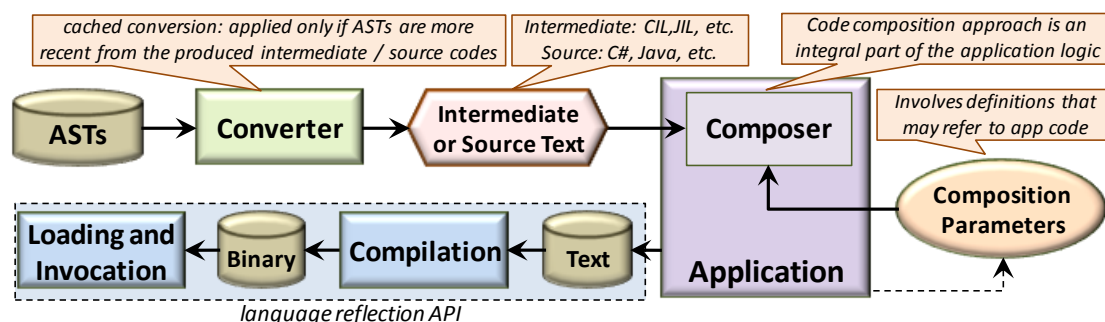


Figure 7.8 – Applying the generative MDE process with runtime staging; the application composes intermediate or source text and deploys the language reflection API for compilation and invocation (JIL stands for Java Intermediate Language, CIL for the Common Intermediate Language of .NET).

7.3.3 Self MDE Deployment

MDE is a widely used software engineering approach but is typically practiced separately from the rest of the development process. Generative MDE tools are used to transform model entities to source code that is then incorporated into an integrated development environment (IDE) for further processing and linking with the remaining application code. In this sense, MDE requires third party tools that cannot always be properly integrated in the deployed IDE. Of course, there are exceptions like Eclipse which provides typical development support (e.g. for Java) while also integrating various modeling frameworks (e.g. Eclipse Modeling Framework [Eclipse03]) through plugins. However, in such cases the integration of the modeling support is usually language-dependent while the adoption of any additional modeling framework for which no IDE plugin exists (e.g. legacy tools) still requires the model authoring and code generation to be separated from the main development process. The problem escalates when dealing with large-scale applications that may involve multiple models authored and maintained by different MDE tools. In this case, developers should be aware of all deployed models, their associated tools as well as the source locations they affect. Since such information exists only as developers' knowledge and is typically not documented somewhere, a large number of deployed models may lead to severe organization and maintenance issues.

Nevertheless, when using generative MDE tools the target is always to obtain the generated code; the MDE tool is typically not launched again unless the model needs to be updated, while any model updates result in the model code being regenerated and then linked again with normal program code. Towards this direction, we try to bring the MDE deployment as close as possible to the actual application development by adopting metaprogramming practices to orchestrate the MDE deployment directly within the program source. Essentially, we extend our proposition regarding staged model-driven generators to include initial stage execution code responsible for launching the MDE tool. In this sense, model editing is now performed *online* during the compilation of the main program instead of being a *separate offline process*. Then, the process continues as before, with the generator macros loading the modeled entities stored in AST form, inserting them into the program source and finally translating them along with the normal program code to deliver the final executable. The extended proposition is illustrated in Figure 7.9.

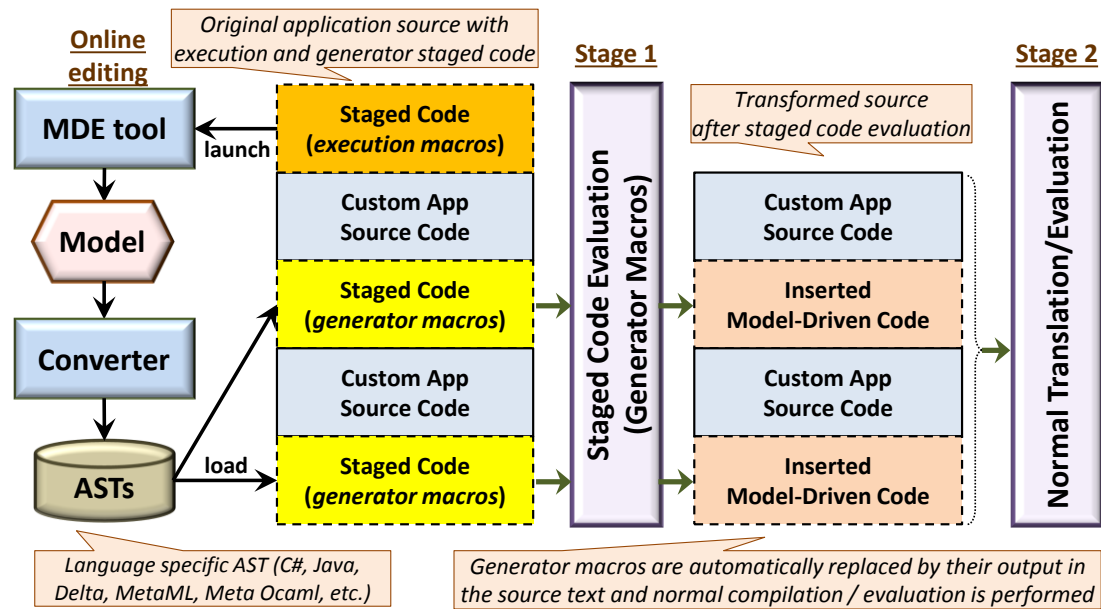


Figure 7.9 – Extending the staged model-driven generator proposition to offer self MDE deployment.

The staged code for launching the model and the staged code for inserting the model code are both parts of the same executing staged metaprogram and are separated only semantically. Practically, we suggest that the initial code responsible for launching the MDE tools will block the stage execution until all model updates have been performed and the model code is converted to AST form. Thus, when stage execution continues with the subsequent generator code, the ASTs they receive as input will be already updated, thus reflecting the latest model changes.

Applying this methodology, the entire knowledge regarding the deployment of MDE tools becomes explicit data, specified within the program itself. This may be particularly important when multiple MDE tools are used to generate code for several parts in the same application. In such cases, source code originating from multiple tools may cause confusion as to which tool produced each code segment and for what purpose. Thus, having a clear association between the tool and the model code it generates directly visible within the program source can be a significant aid towards understanding the purpose and the linkage of the generated code segments within the overall application.

Deploying the proposed approach in a context where a programmer is also an MDE tool expert is straightforward; however in a collaborative development setup where programmers and model developers have distinct responsibilities it requires additional

consideration. Initially, programmers implement the metaprogram containing the execution macros for externally launching the MDE tool and the generator macros for loading the model code in AST form. These sources are given to the model developers that are responsible for interacting with the launched MDE tools during staged execution in order to generate the model entities. The latter implies that model developers also have source code tools available to them so as to compile the sources, however does not require any additional programming knowledge. The resulting model is then converted to AST form and given back to programmers. Any source code changes should not affect the execution macros to allow model developers launch the MDE tools and work in parallel, while any new model updates simply result in updated AST versions that are supplied to programmers.

For programmers that have no MDE tools installed, the staged execution should skip the first step of the process and continue directly with the insertion of the model code using already existing AST data, created and supplied at an earlier point by a model developer. This functionality can be easily abstracted within the initial execution macros; the macros can check if the required MDE tools are available or not and act accordingly. Additionally, regardless of the presence of the MDE tools, it is possible to interactively ask users if they want to perform any model editing (e.g. through a simple dialog); if they choose not to edit any model, there is no MDE tool invocation and the generators just use the previous AST versions.

7.3.4 Case Studies

To validate our approach we have carried out three case studies with proof-of-concept prototypes, one focusing on user-interface code generation, another one creating an entire class hierarchy based on a given model and a third one combining the two previous models into a single application. The goals of our studies were: (i) to show that the maintenance issues are effectively eliminated; (ii) to highlight the flexibility of inserting the generated model code in-place along with the custom application code using generative macros; and (iii) to show that the MDE tool chain can be successfully incorporated as part of the metaprogram.

7.3.4.1 User-Interface Generation

We have adopted the *wxFormBuilder* [Hurtado], a popular publicly available interface builder for the *wxWidgets* cross-platform library that generates interface descriptions into a custom language-neutral format called XRC (XML Interface Resources). Using this tool, we constructed a simple graphics painting application, the latter actually practiced in alternative ways, such as with single authoring project or alternatively with multiple independent projects (i.e. multiple XRC models). This way we could also assert the compositional flexibility of our proposed approach in combining independently authored interfaces under a single system. Then we build an appropriate converter to transform the XRC data into Delta language ASTs. Finally, using the metaprogramming features of the Delta language we imported and manipulated the application ASTs, and also added extra interactive features and behavior to it, besides the ones introduced only with the *wxFormBuilder*. In-between this process we reloaded the visual models and regenerated the XRC files many times, to test that no maintenance issues arise by this cycle.

Figure 7.10 illustrates one of the implemented user-interface composition scenarios based on two separate interface descriptions. The toolbar of the second interface is initially retrieved by cropping its top level frame, and is then inserted directly in the top level frame of the paint application. Finally, the combined interface is produced by inlining the transformed paint application AST.

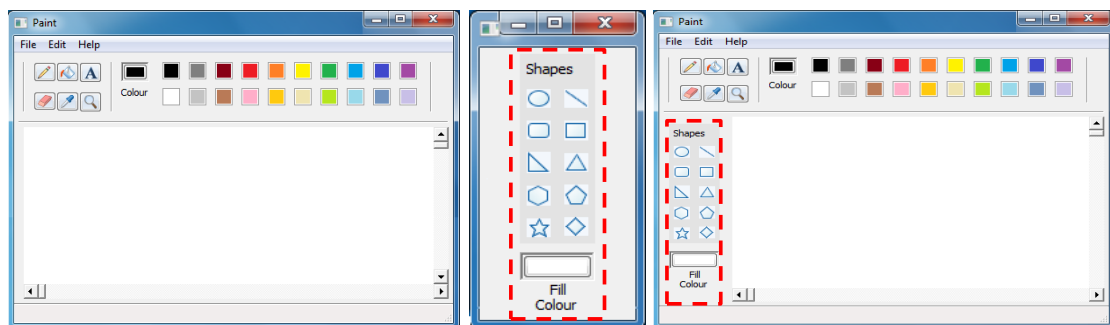


Figure 7.10 – Examples of generated interfaces: *Left*: Original application GUI authored by the interface builder; *Middle*: Custom toolbar authored as a separate interface; *Right*: Composing the two previous interfaces through AST manipulation.

Additionally, we assessed the self MDE deployment proposition by launching the *wxFormBuilder* directly from the metaprogram to allow interactive editing of the user interface during compilation. The latter is accomplished with the following code:

```

&modelProjectPath = "/models/paint.fbp";           ← wxFormBuilder project file
&std::fileexecute("start wxFormBuilder" + modelProjectPath);
...the above call suspends execution until model editing is complete and wxFormBuilder is closed...
&modelProject = xml::load(modelProjectPath);
&model = load_xrc(modelProject.path); ← loads the updated XRC for the Paint GUI
&ast = Converter::xrc2ast(model);           ← convert the XRC data to AST
!(ast);                                     ← insert the model code into the program source

```

7.3.4.2 Class Hierarchy Generation

We used the Eclipse Modeling Framework [Eclipse03] to model a class hierarchy for the development of a paint application toolset. The hierarchy contained the abstract notion of shapes, as well as concrete drawable shapes like points, lines, circles, etc. The model was created through the *Ecore* meta-model and its specification was generated in XMI format. We also built a converter to transform XMI data to Delta language ASTs.

Figure 7.11 shows the model, the generated AST (shown as source code for clarity) as well as the deployment code required to inline the code AST in-place with the normal program code. Again during the process, we reloaded the model and regenerated the XMI specification to verify that no maintenance issues were introduced in the development process.

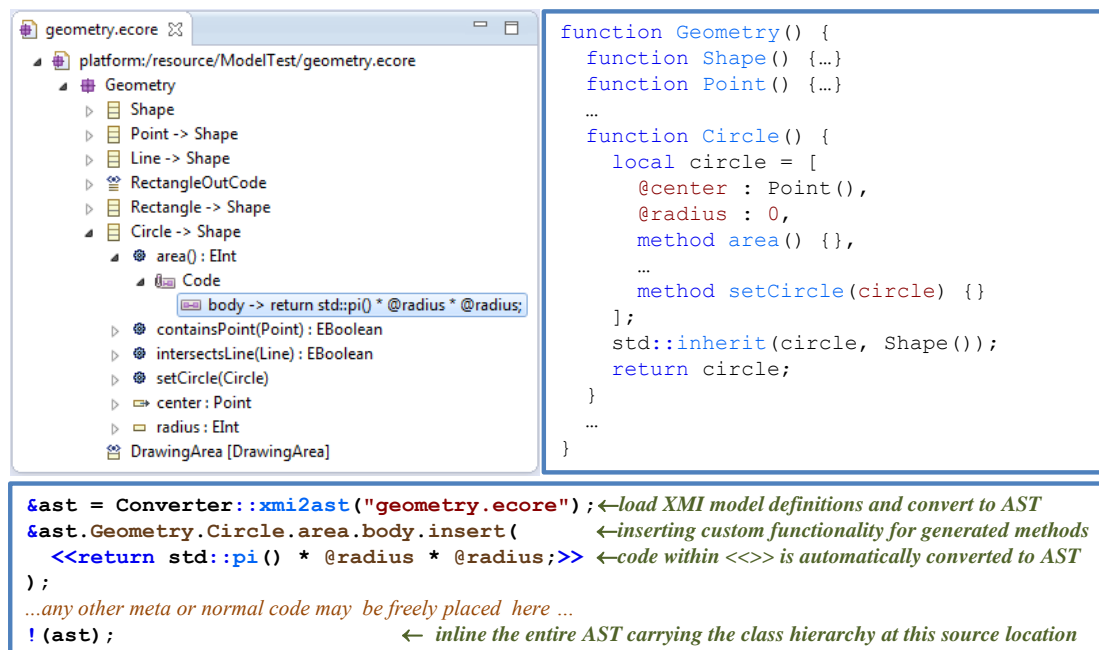


Figure 7.11 – Top-left: Ecore model of the target class hierarchy; Top-right: Code structure (AST) generated by the model; Bottom: Deployment code for loading and converting the model to AST, performing manual updates through AST editing and inlining the final AST code.

To implement the methods of the modeled classes we practiced two alternative approaches: (i) we specified the code directly in the model through the use of special *EAnnotation* elements (Figure 7.11 top-left, highlighted); and (ii) we inserted the code through AST editing as part of the metaprogram (Figure 7.11 bottom, 2nd statement). The second approach may seem more difficult to adopt, but in fact it is easy to develop and offers several advantages over the first one. In the first approach, code is entered as raw text providing no programming facilities or any potential for parameterization or reuse. On the contrary, in the second approach, code is created through quasi-quotes at a syntactic level offering facilities like highlighting, auto-completion, refactoring tools, etc. Additionally, the representation of code as ASTs allows adopting standard software engineering practices like parameterization, encapsulation and modular composition. The only issue related to programmatically extending the generated AST is the need for traversing the AST to locate the nodes that require extensions, something requiring knowledge of the code generation scheme as well as internal AST information. Nevertheless, this can be augmented by an AST decoration process that offers direct navigation across AST nodes using the named entities of the class hierarchy, as illustrated by the access of a particular method body as `ast.Geometry.Circle.area.body` (Figure 7.11 bottom, 2nd statement). This way, knowledge of the model entities and a simple tree manipulation API are sufficient for a developer to introduce elaborate AST extensions.

The deployment of the Eclipse Ecore model editor was practiced either as a separate external tool, or as part of the main program compilation, launching it during stage code execution. Additionally, we explored the alternative of implementing the model editor as an inherent part of the metaprogram, i.e. without launching any external applications. In particular, we built a simple GUI offering an editable tree control to specify the class hierarchy, effectively emulating the Ecore model editor functionality. Such an approach may take advantage of executing in the same address space with the metaprogram that will utilize the model output, for example storing the generated ASTs directly in a metaprogram variable accessible from the generator macros, thus minimizing the overhead of storing and reloading the AST data. Also, such a custom editor need not be implemented from scratch for any program; it can be implemented once as a reusable compile-time library and then deployed anytime a program requires self MDE deployment through the particular editor.

7.3.4.3 Combining Multiple MDE Tools

As a last case study, we focused on combining multiple MDE tools to generate code for a single application. In particular, we used the previously discussed user-interface model for the painting application along with the class hierarchy model used to implement the core logic of the paint application. In this context, a simple concatenation of the generated source code caused no direct compilation conflicts; however it was far from sufficient for deriving a fully-functional application. In fact, multiple manual updates were necessary involving both generated components and requiring bidirectional dependencies. Firstly, the event handling code required knowledge of the separately generated implementation classes. Then, certain methods of the class hierarchy like draw required invoking UI-related operations. However, the class hierarchy model was unaware of the deployed UI library, meaning that such information could not be available in the model and would thus have to be explicitly expressed as a manual extension in the generated sources. Finally, we needed to combine the generated code with the custom application logic. The meta-code implementing this functionality is outlined below with details removed for clarity.

```
using wx;           ← normal code, directive for importing the wxWidgets GUI toolkit
&paintUI = nil;    ← meta-code variable, to store the AST of the paint application UI code
&shapesUI = nil;    ← meta-code variable, to store the AST of the shapes toolbar UI code
&classes = nil;    ← meta-code variable, to store the AST of the class hierarchy for the toolset
&{                ← an entire block of meta-code begins here
paintUI = Convert::xrc2ast("paint.xrc"); ← load XRC UI data and convert to AST
shapesUI= Convert::xrc2ast("toolbarShapes.xrc");
classes = Convert::xmi2ast("paint.ecore"); ← load XMI model and convert to AST
Tree::Crop(shapesUI, "shapes"); ← drop the outer frame inserted by wxFormBuilder
canvas = Tree::Get(paintUI, "canvas"); ← get the code creating the canvas paint panel
Tree::InsertBefore(paintUI, shapesUI, canvas); ← insert the code for the shapes
toolbar into the paintUI frame, placing it before the code of the canvas
classes.Geometry.Circle.draw.body.insert( ← implementation for Circle::draw(dc)
    <<dc.drawcircle(@center, @radius);>>); ← dc: arg, @center and @radius: attributes
...other shape method implementations are specified here as well...
Tree::Insert(paintUI,          ← insert an application event handler for circle shape button
    "circle", "EVT_COMMAND_BUTTON_CLICKED",
    <<Paint.SetSelectedTool("shapeCircle");>> ← handler code specified as an AST
);
...other event handlers are inserted here as well...
}                               ← the block of meta-code ends here
...any other meta or normal code may be freely placed here...
!(classes);                    ← inline the entire AST carried by classes at this source location
...any other meta or normal code may be freely placed here...
!(paintUI);                    ← inline the entire AST carried by paintUI at this source location
...any other meta or normal code may be freely placed here...
```

The option for self MDE deployment was explored here as well. In particular, we included as part of the metaprogram data information about the models deployed in the application (e.g. model type, modeling tool and execution path) and used this information to automatically assemble a dialog enabling the interactive launching of all MDE tools directly from the execution context of the metaprogram.

Chapter 8

Conclusions and Future Work

“Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience.”

-Roger Bacon

8.1 Summary

In this thesis we discussed a systematic proposition for *Integrated Metaprogramming Systems* covering aspects of language, programming model, tool support and deployment practices. Our primary motivation was the need for a methodological integration between metaprogramming and normal programming, as we consider impractical to have diverse development styles and approaches amongst the two universes. Since metaprograms are essentially programs, we identified a set of prominent requirements for achieving such integration, effectively enabling metaprograms to directly adopt the engineering practices, processes and tools of normal programs. We consider the latter to represent a paradigm shift towards an integrated code of practice where metaprograms are no longer considered to belong to a segregated and customized language domain.

Central to our proposition is the notion of integrated metaprograms resulting by the collection of all code fragments of the same stage nesting, following their order of appearance in the main source, while implying a lexically-scoped control flow. We proved that the integrated model is at least as expressive as the traditional stage evaluation in existing multi-stage languages, while it provides significant advantages from a software engineering perspective. The latter was demonstrated through the detailed case studies that included demanding scenarios of exception handling, design patterns and design by contract.

Apart from integrating metaprogramming with normal programming at the language level, we also focused on integrating it at the tool chain level, turning metaprograms to first-class citizens of the programming environments. Metaprograms, as well as the

transformations they perform on normal programs, are both included in the workspace manager thus facilitating source code review and enabling a full-scale meta-build process with informative error reporting and source-level debugging. The meta-build process is achieved by supporting for metaprograms all build flags and properties of normal programs. Additionally, a typical recursive build process is applied for any of their dependencies prior to actual compilation. Then comprehensive error reporting is offered by tracing back the entire chain of source locations involved in generating an erroneous code fragment. Finally, full-scale source-level debugging is supported by automatically generating metaprogram breakpoints from original source breakpoints, and by instrumenting the language compiler with the debugger backend so that debugging sessions for evaluated stages can be initiated.

Once effectively integrated normal programs and metaprograms in terms of language and tools, we explored how certain paradigms and best practices of normal programming may be extended to directly apply in a metaprogramming context. In this direction, we focused on the application of AOP in the context of metaprogramming and introduced aspect-orientation in the entire processing pipeline of a multi-stage language. In particular, we identified three aspect categories: (i) *pre-staging aspects*, applied on the original source code in order to introduce staging or transform existing stages; (ii) *in-staging aspects* applied on stage metaprograms to apply typical AOP on stage code; and (iii) *post-staging aspects*, applied in the outcome of the staging process (compile-time metaprogramming only) to perform traditional AOP transformations. In particular, aspect deployment has been practiced through a model treating aspects as batches of transformation programs written in the same language. This model fits well with typical multi-stage metaprogramming practices, allows exploiting all discussed metaprogramming facilities of the IDE while it supports the opposite direction, that of deploying metaprogramming for aspects.

Finally, we focused on deriving a code of practice that can exploit metaprogramming to achieve reusability at a software design level not currently feasible with the available reusability constructs of existing languages. More specifically, we discussed the delivery of reusable design pattern implementations built through metaprogramming. In this approach, a metaprogram encompassing the pattern logic serves as a pattern generator while the details of the particular application context are

defined through pattern instantiation parameters. The latter methodologically resembles the way *class templates* are defined and used in C++ and it essentially leads to *pattern templates* being more macroscopic reusable artifacts. Then we presented a methodology for implementing combinations of exception handling policies as reusable libraries. In this case, we encapsulated the policy logic into meta-functions that are invoked with appropriate configuration parameters which control the way the generated exception handling source code is structured. Finally, we deployed metaprogramming to cope with the maintenance issues arising from the usage of generative MDE tools. In particular, we proposed an improved model-driven code of practice where the generator components of MDE tools output source fragments as ASTs. Then, application source code encompasses metaprograms to load and compose such ASTs as required, and to finally generate code at the required source locations using generative directives.

8.2 Conclusions

Throughout the entire thesis we have emphasized two primary arguments driving our research work: (i) metaprograms, being programs as such, deserve all programming features, practices and tools available to normal programs; and (ii) essential metaprogram development can be only achieved with the availability of the required tools across the entire development cycle.

During the initial design phases of our integrated metalanguage we mostly focused on the code generation directives (i.e., the inline staging tag). While this allowed expressing various metaprogramming scenarios, we quickly observed that it suffered from a practical perspective. More specifically, certain common expressions had to be repeated with every inline directive since, syntactically, generative directives are expressions, and thus disabled the modular encapsulation of reusable definitions such as functions or classes. Additionally, the need to have some kind of state propagated from one directive to another appeared very frequently. In this context, we required a way to declare shared variables or instantiate shared objects, something again impossible with the typical generative expressions. As a result, we came out with the idea of introducing staged code that merely incorporates such required definitions, becoming syntactically visible to any following generative directives. Then, we also realized that besides a collection of generative directives, a metaprogram may reflect

more comprehensive algorithmic characteristics, implying control flow scenarios far beyond the mere sequence of generative calls. To address this demand we allowed statements to be part of staged definitions. Eventually, all the previous have led to the introduction of the *execute* staged tag which helped to better consolidate and support the idea of combining related staged fragments into a coherent integrated program.

Even from the very early phases of this thesis, involving the writing of non-trivial metaprograms in various languages, it became obvious that metaprogramming was becoming a tedious and unconventional task with the absence of development tools. Initially, we experienced difficulties in tracing compile errors caused by defects in the logic of staged code due to lack of elaborate information on the root of the error. Then, runtime errors in stages had to be resolved, something that proved to be an even harder task due to absence of any debugging instruments. In fact, the latter was a little surprising since we had to deploy very old methods with extra diagnostic messages in the original source code. Practically, this programming experience was a sort of paradigm mismatch. On the one hand we applied an advanced programming method, and on the other hand we were bound in toolsets so primitives as two decades ago. Needless to mention that similar issues arose as the size of the metaprograms escalated, since soon we started to notice the absence of editing automations and build facilities for metaprograms.

Concerning aspects, it should be noted that they were originally targeted only on the final program, aiming to allow arbitrary algorithmic transformations, as opposed to simple pattern matching. The latter was fully supported by our custom AOP implementation treating aspects as non-staged transformation metaprograms. Now, quickly after the notion of integrated metaprograms was formulated, where we revisited stages as full-scale programs, we realized there was no reason to exclude stages from the automation of crosscutting concerns. This not only required an effective application of aspects on metaprograms, but also challenged a uniform approach applicable in both worlds. This led to an exhaustive analysis of all potential points in which aspects might be required, eventually leading to the triplet of *pre*-, *in*-, and *post*- staging aspects. The latter involved an early study showing the way such stage aspects can be accommodated in languages with runtime staging by exploiting reflection mechanisms.

The proposition for an alternative aspect system relying on transformation batches came actually as a spin-off while trying to validate with concrete case studies our notion of aspects for stages. Very early we started considering aspects to constitute a special case of non-staged metaprograms, delivered with custom languages for automating query, matching and transformation. Since our work was driven by an integration discipline on metaprograms, we thought that the necessity for a separate language just for aspects was somehow a barrier. Hence, we focused on turning all aspect automations to a library, setting aspect application as a build preprocessing stage, and treating aspects as non-staged metaprograms with no separate language.

In this case, after implementing case studies, we have three surprising observations. Firstly, this approach allowed aspects to be treated as normal programs, offering all features of the development environment, ranging from debugging, to build system and source control. Secondly, it allowed deploying staged metaprograms inside aspects, thus not only offering aspects for stages, but also bringing stages to aspects, something we never initially imagined. At this point we assume this to be a good sign for the completeness of the method. Finally, we understood that we may introduce aspects on aspects as well, semantically resembling nested staged metaprograms, something we considered to gracefully close the circle.

In the context of deployment practices, we investigated numerous scenarios where metaprogramming can help accomplish better reuse. In particular, design pattern generators and exception handling templates demonstrate the reuse power of integrated metaprograms beyond what is possible with the reusability constructs of existing languages. An in-depth case study concerned the proposition for staged model-driven generators to alleviate the serious maintenance issues inherent in the manual modification of generated model code. Besides the various technical advantages of the approach, we eventually received a more general message. In particular, we started thinking that in any of the current practices, tools and processes, wherever code generation is somehow involved, we should probably explore potential improvements by metaprogramming solutions.

Overall, this thesis focused on deriving a systematic discipline for developing and deploying metaprograms, directly driven by software engineering requirements. We consider metaprogramming to play an increasingly important role in the software

development process. As software systems continue to grow in size and complexity, reuse should shift towards more flexible and powerful solutions, with metaprograms being a very promising solution. In this context, we strongly emphasize the criticality of a demand-driven perspective and the need to support large-scale and real-life adoption of metaprograms. We consider our work to be step ahead in this direction.

8.3 *Future Work*

In this thesis we focused on the most prominent of the identified requirements, while some of the areas remain open and require additional research work. Below, we briefly discuss key topics for future work.

One of the identified issues, being beyond the scope of this thesis, concerns the provision of source editing automations for metaprograms, in particular auto-completion, quick information, parameter help, and go-to definition. Now, for typed non-meta languages such automations rely on the creation of symbol tables during editing, usually involving some sort of custom parsing of the edited files, as well as of those imported. In case of either untyped or dynamically-typed non-meta languages the general problem is still open since it requires editing-time evaluation.

Regarding metalanguages, stage evaluation is apparently required to make sure that the editing tools process the actual resulting program. More specifically, for any stage or the final program the evaluation of any inner stages is needed to obtain the resulting source code. Generally, this case is similar to the previous one, as it also requires editing-time evaluation to derive information for program elements, and requires further research to provide efficient and scalable solutions. However, we assume the problem to be more challenging in the context on untyped metalanguages due to the combined complexity. In any case, since editing automations are currently a necessary element of all existing development tools, future work in this domain becomes absolutely critical.

Another topic for future research work regarding metaprogramming tools concerns advanced editing views. Currently, we extract stage metaprograms and include them as separate read-only source files inside the workspace to help in the administration and understanding of stages. However, editing stage code is still bounded to the context of the original source file. Since the latter likely encompasses a lot of source

code that is irrelevant to the staging logic it unnecessarily overloads the stage programming task. In this direction, specific editing facilities to improve the task are needed, such as folding code not belonging to the current stage, enabling programmers also experience a visually integrated program. Additionally, layered editing views for stages and the transformations they introduce are required, enabling seamless navigation in the entire transformation path of code fragments from the original source to their eventual form in the final source. Such views may support editing directly on the final program by translating back to respective modifications in the context of the original source file.

Regarding metaprogram debugging, we envision more advanced facilities that are optimized towards AST inspection tasks. In our work we discussed typical facilities like call stack, watches, breakpoints and execution tracing that should be supported under metaprogram debugging. In this context, we consider that new debugging facilities are required optimized for inspecting the generative behavior of metaprograms. For instance, appropriate visualizations of the various transformed versions of the main program AST, resulting from stages, may be very helpful. In our work we supported visualizations of inspected AST values. Such a feature can be further improved with more interactive features such as: folding or unfolding sub-trees, search using language constructs, display of unparsed code for selected nodes, node bookmarking, configurable visualization approaches, etc.

Finally, the implementation of integrated metaprogramming systems for typed languages such as C++, C# and Java is a very challenging endeavor. We already investigated a few examples on a hypothetical meta-C++ assuming an integrated metaprogramming model. In fact, all metaprogramming tool extensions we proposed have been designed without a particular dependency on the typing approach of the language or its staging model. Nevertheless, a typed language universe may introduce additional requirements not apparent in an untyped language context, potentially requiring further investigation.

Overall, we consider that a full-scale implementation of our approach in a mainstream and popular typed language will not only make the advantages of integrated metaprogramming far more evident, but it will genuinely push forward the art of metaprogramming and lead to more advanced programming practices in the future.

Bibliography

- [Abrahams] David Abrahams and Aleksey Gurtovoy. 2004. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional.
- [Alexandrescu] Andrei Alexandrescu. 2010. *The D Programming Language*. Addison-Wesley Professional.
- [Arnold] Ken Arnold, James Gosling and David Holmes. 2005. *The Java Programming Language*. Fourth Edition, Addison-Wesley Professional.
- [Arnout] Karine Arnout. 2004. *From Patterns to Components*. Ph.D. Thesis, Dept. Computer Sciences, Chair of Software Engineering, ETH Zurich. Available at: http://se.inf.ethz.ch/old/people/arnout/patterns/download/karine_arnout_phd_thesis.pdf. Accessed 11/2013.
- [Bachrach99] Jonathan Bachrach and Keith Playford. 1999. *D-expressions: Lisp Power, Dylan Style*. Available at: <http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf>. Accessed 11/2013.
- [Bachrach01] Jonthan Bachrach and Keith Playford. 2001. *The Java syntactic extender (JSE)*. In Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '01). ACM, New York, NY, USA, pp. 31-42. Available at: <http://doi.acm.org/10.1145/504282.504285>.
- [Baker] Jason Baker and Wilson Hsieh. 2002. *Runtime aspect weaving through metaprogramming*. In Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02). ACM, New York, USA, pp. 86-95. Available at: <http://doi.acm.org/10.1145/508386.508396>.
- [Batory] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. 1998. *JTS: Tools for Implementing Domain-Specific Languages*. In Proceedings of the 5th International Conference on Software Reuse (ICSR '98). IEEE Computer Society, Washington, DC, USA, pp.143-153. Available at: <http://dx.doi.org/10.1109/ICSR.1998.685739>.
- [Bawden] Alan Bawden. 1999. *Quasiquote in Lisp*. In Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pp. 88–99, San Antonio. University of Aarhus, CS Department. Invited talk. Available at: <http://repository.readscheme.org/ftp/papers/pepm99/bawden.pdf>. Accessed 11/2013.
- [BCEL] Apache Commons. 2006. *BCEL - Byte Code Engineering Library*. <http://commons.apache.org/bcel/manual.html>. Accessed 11/2013.
- [Becker] Pete Becker. 2011. *C++ International Standard*. Available at: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf>. Accessed 11/2013.
- [Bravenboer] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. *Stratego/XT 0.17. A language and toolset for program*

- transformation*. Sci. Comput. Program. 72, 1-2 (June 2008), pp. 52-70. Available at: <http://dx.doi.org/10.1016/j.scico.2007.11.003>.
- [Bright] Walter Bright. 2010. *The X Macro*. <http://www.drdobbs.com/cpp/the-x-macro/228700289>. Accessed 11/2013.
- [Bruneton] Eric Bruneton. 2007. *ASM 4.0 A Java bytecode engineering library*. 2007. Available at: <http://download.forge.objectweb.org/asm/asm4-guide.pdf>. Accessed 11/2013.
- [Bryant] Avi Bryant and Robert Feldt. 2002. *AspectR - Simple aspect-oriented programming in Ruby*. <http://aspectr.sourceforge.net/>. Accessed 11/2013.
- [Burmako] Eugene Burmako. 2013. *Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming*. In Proceedings of the 4th Workshop on Scala (SCALA '13). ACM, New York, NY, USA, Article 3, pp.1-10, 10 pages. Available at: <http://doi.acm.org/10.1145/2489837.2489840>.
- [Cacho] Néelio Cacho, Thaís Batista and Fabrício Fernandes. 2005. *AspectLua - A Dynamic AOP approach*. In Journal of Universal Computer Society, 11(7):1177-1197. Available at: <http://dx.doi.org/10.3217/jucs-011-07-1177>.
- [Calcagno01] Cristiano Calcagno, Walid Taha, Liwen Huang and Xavier Leroy. 2001. *A bytecode-compiled, type-safe, multi-stage language*. Available at: <http://www.cs.rice.edu/~taha/publications/preprints/pldi02-pre.pdf>. Accessed 11/2013.
- [Calcagno03] Cristiano Calcagno, Walid Taha, Liwen Huang and Xavier Leroy. 2003. *Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection*. In Proceedings of the Second International Conference on Generative Programming and Component Engineering (GPCE 2003), Erfurt, Germany, September 22-25, Springer LNCS 2830, pp. 57-76. Available at: http://dx.doi.org/10.1007/978-3-540-39815-8_4.
- [Chiba] Shigeru Chiba. 1995. *A metaobject protocol for C++*. In Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA '95). ACM, New York, NY, USA, pp. 285-299. Available at: <http://doi.acm.org/10.1145/217838.217868>.
- [Coelho08A] Roberta Coelho, Awais Rashid, Alessandro Garcia, Fabiano Ferrari, Nélio Cacho, Uirá Kulesza, Arndt von Staa and Carlos Lucena. 2008. *Assessing the Impact of Aspects on Exception Flows: An Exploratory Study*. In Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008), Paphos, Cyprus, July 7-11, Springer LNCS 5142, pp. 207-234. Available at: http://dx.doi.org/10.1007/978-3-540-70592-5_10.
- [Coelho08B] Roberta Coelho, Awais Rashid, Arndt von Staa, James Noble, Uirá Kulesza, and Carlos Lucena. 2008. *A catalogue of bug patterns for exception handling in aspect-oriented programs*. In Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP '08). ACM,

- New York, NY, USA, Article 23, pp. 1-13. Available at: <http://doi.acm.org/10.1145/1753196.1753224>.
- [Cole] Leonardo Cole and Paulo Borba. 2005. *Deriving refactorings for AspectJ*. In Proceedings of the 4th international conference on Aspect-oriented software development (AOSD '05). ACM, New York, NY, USA, Chicago, Illinois, pp. 123-134. Available at: <http://doi.acm.org/10.1145/1052898.1052909>.
- [Costanza] Pascal Costanza. 2004. *A short overview of AspectL*. In Proceedings of the European interactive workshop on aspects in software (Berlin, Germany, September 23-24, 2004). EIWAS. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.5235&rep=rep1&type=pdf>. Accessed 11/2013.
- [Czarnecki] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz and Walid Taha. 2004. *DSL implementation in MetaOCaml, Template Haskell, and C++*. In Domain-Specific Program Generation, Springer LNCS 3016, pp. 51-72. Available at: http://dx.doi.org/10.1007/978-3-540-25935-0_4.
- [Descent] *Descent: An Eclipse plugin providing an IDE for the D programming language*. <http://www.dsource.org/projects/descent>. Accessed 11/2013.
- [Doshi] Gunjan Doshi. 2003. *Best practices for exception handling*. <http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html>. Accessed 11/2013.
- [Draheim] Dirk Draheim, Christof Lutteroth, and Gerald Weber. 2005. *A type system for reflective program generators*. In Proceedings of the 4th international conference on Generative Programming and Component Engineering (GPCE'05), Tallinn, Estonia, September 29 - October 1, Springer LNCS 3676, pp. 327-341. Available at: http://dx.doi.org/10.1007/11561347_22.
- [Dupuy] Emmanuel Dupuy. 2008. *JD Java Decompiler*. <http://jd.benow.ca/>. Accessed 11/2013.
- [Dutchyn] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. 2006. *Semantics and scoping of aspects in higher-order languages*. Sci. Comput. Program. 63, 3 (December 2006), pp. 207-239. Available at: <http://dx.doi.org/10.1016/j.scico.2006.01.003>.
- [Dybvig92] R. Kent Dybvig. 1992. *Writing Hygienic Macros in Scheme with Syntax-Case*. Technical Report, Indiana University Computer Science Department. Available at: <http://www.cs.indiana.edu/~dyb/pubs/tr356.pdf>. Accessed 11/2013.
- [Dybvig09] R. Kent Dybvig. 2009. *The Scheme Programming Language (fourth edition)*. The MIT Press, ISBN 978-0-262-51298-5.
- [Eaddy] Marc Eaddy, Alfred Aho, Weiping Hu, Paddy McDonald, and Julian Burger. 2007. *Debugging aspect-enabled programs*. In Proceedings of the 6th international conference on Software composition (SC'07), Braga, Portugal, March 24-25, Springer LNCS 4829, pp 200-215. Available at: http://dx.doi.org/10.1007/978-3-540-77351-1_17.

- [Eckhardt] Jason Eckhardt, Roumen Kaiabachev, Emir Pašalić, Kedar Swadi, and Walid Taha. 2005. *Implicitly heterogeneous multi-stage programming*. In Proceedings of the 4th international conference on Generative Programming and Component Engineering (GPCE'05), Tallinn, Estonia, September 29 - October 1, Springer LNCS 3676, pp. 275-292. Available at: http://dx.doi.org/10.1007/11561347_19.
- [Eclipse03] The Eclipse Foundation. 2003. *Eclipse Modeling Framework (EMF)*. <http://www.eclipse.org/modeling/emf/> Accessed 11/2013.
- [Eclipse05] The Eclipse Foundation. 2005. *AJDT: AspectJ Development Tools*. <http://www.eclipse.org/ajdt/> Accessed 11/2013.
- [Eich] Brendan Eich. 2005. *JavaScript at ten years*. In Proceedings of the tenth ACM SIGPLAN international conference on Functional programming (ICFP '05). ACM, New York, USA, pp. 129-129. Available at: <http://dx.doi.org/10.1145/1090189.1086382>.
- [Engler] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 1996. *'C: a language for high-level, efficient, and machine-independent dynamic code generation*. In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96). ACM, New York, NY, USA, pp. 131-144. Available at: <http://doi.acm.org/10.1145/237721.237765>.
- [Erdweg] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. *SugarJ: library-based syntactic language extensibility*. In Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11). ACM, New York, NY, USA, pp. 391-406. Available at: <http://doi.acm.org/10.1145/2048066.2048099>.
- [Fähndrich] Manuel Fähndrich, Michael Carbin, and James R. Larus. 2006. *Reflective program generation with patterns*. In Proceedings of the 5th international conference on Generative programming and component engineering (GPCE '06). ACM, New York, NY, USA, pp. 275-284. Available at: <http://doi.acm.org/10.1145/1173706.1173748>.
- [Filho06a] Fernando Castor Filho, Cecília Mary F. Rubira, Raquel de A. Maranhão Ferreira and Alessandro Garcia. 2006. *Aspectizing Exception Handling: A Quantitative Study*. Advanced Topics in Exception Handling Techniques. Springer LNCS 4119, pp. 255-274. Available at: http://dx.doi.org/10.1007/978-3-540-73589-2_9.
- [Filho06b] Fernando Castor Filho, Nélío Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília M. F. Rubira. 2006. *Exceptions and aspects: the devil is in the details*. In Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14), ACM, New York, NY, USA, Portland, Oregon, November 5-11, pp. 152-162. Available at: http://dx.doi.org/10.1007/11818502_14.
- [Filho07] Fernando Castor Filho, Alessandro Garcia and Cecília M. F. Rubira. 2007. *Extracting Error Handling to Aspects: A Cookbook*. In Proceedings of the 23rd IEEE International Conference on Software

- Maintenance (ICSM 2007), Paris, France, October 2-5, pp. 134-143. Available at: <http://dx.doi.org/10.1109/ICSM.2007.4362626>.
- [Filho09] Fernando Castor Filho, Nélío Cacho, Eduardo Figueiredo, Alessandro Garcia, Cecília M. F. Rubira, Jefferson Silva de Amorim and Hítalo Oliveira da Silva. 2009. *On the modularization and reuse of exception handling with aspects*. In Software: Practice and Experience, Volume 39, Issue 17, pp. 1377-1417, 10 December 2009. Available at: <http://dx.doi.org/10.1002/spe.939>.
- [Findler] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. *DrScheme: a programming environment for Scheme*. J. Funct. Program. 12, 2 (March 2002), pp. 159-182. Available at: <http://dx.doi.org/10.1017/S0956796801004208>.
- [Fleutot07a] Fabien Fleutot. 2007. *Metalua Manual*. <http://metalua.luaforge.net/metalua-manual.html>. Accessed 11/2013.
- [Fleutot07b] Fabien Fleutot and Laurence Tratt. 2007. *Contrasting compile-time meta-programming in Metalua and Converge*. In Proceedings of Workshop on Dynamic Languages and Applications. Available at: http://tratt.net/laurie/research/pubs/papers/fleutot_tratt_contrasting_compile_time_meta_programming_in_metalua_and_converge.pdf. Accessed 11/2013.
- [Franz] Franz Inc. 2012. Allegro CL 9.0. <http://www.franz.com/products/allegrocl/> Accessed 11/2013.
- [Gamma] Erich Gamma, Ralph Johnson, John Vlissides and Richard Helm. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Professional. ISBN 978-0201633610.
- [Ganz] Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. *Macros as multi-stage computations: type-safe, generative, binding macros in MacroML*. In Proceedings of the sixth ACM SIGPLAN international conference on Functional programming (ICFP '01). ACM, New York, NY, USA, pp. 74-85. Available at: <http://doi.acm.org/10.1145/507635.507646>.
- [Garcia] Giovanni Azua Garcia. 2009. *PerfectJPattern project*. <http://perfectjpattern.sourceforge.net/>. Accessed 11/2013.
- [Glück95] Robert Glück and Jesper Jørgensen. 1995. *Efficient multi-level generating extensions for program specialization*. In Proceedings of the 7th International Symposium, PLILP '95 Utrecht, The Netherlands, September 20-22, Springer LNCS 982, pp. 261-272. Available at: http://dx.doi.org/10.1007/3-540-62064-8_22.
- [Glück96] Robert Glück and Jesper Jørgensen. 1996. *Fast Binding-Time Analysis for Multi-Level Specialization*. In Proceedings of the Second International Andrei Ershov Memorial Conference on Perspectives of System Informatics, Novosibirsk, Russia, June 25-28. Springer LNCS 1181, pp. 261-272. Available at: http://dx.doi.org/10.1007/3-540-62064-8_22.

- [Goodenough] John B. Goodenough. 1975. *Exception handling: issues and a proposed notation*. Commun. ACM 18, 12 (December 1975), pp. 683-696. Available at: <http://doi.acm.org/10.1145/361227.361230>.
- [Greenwood] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Cláudio Sant'Anna, Sergio Soares, Paulo Borba, Uirá Kulesza, and Awais Rashid. 2007. *On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study*. In Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP 2007), Berlin, Germany, June 30 - August 3, Springer LNCS 4609, pp. 176-200. Available at: http://dx.doi.org/10.1007/978-3-540-73589-2_9.
- [Heering] Jan Heering, P. R. H. Hendriks, Paul Klint, and J. Rekers. 1989. *The syntax definition formalism SDF - reference manual*. SIGPLAN Not. 24,11 pp. 43-75. Available at: <http://doi.acm.org/10.1145/71605.71607>.
- [Hejlsberg] Anders Hejlsberg, Scott Wiltamuth and Peter Golde. 2006. *The C# programming language*. Addison-Wesley Professional.
- [Hirschfeld] Robert Hirschfeld. 2002. *AspectS - Aspect-Oriented Programming with Squeak*. In Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World (NODE '02), Erfurt, Germany, October 7-10, Springer LNCS 2591, pp. 216-232. Available at: http://dx.doi.org/10.1007/3-540-36557-5_17.
- [Hirzel] Martin Hirzel and Bugra Gedik. 2012. *Streams that compose using macros that oblige*. In Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation (PEPM '12). ACM, New York, NY, USA, pp. 141-150. Available at: <http://doi.acm.org/10.1145/2103746.2103772>.
- [Hongbo] Zhang, Hongbo, and Steve Zdancewic. 2013. *Fan: compile-time metaprogramming for OCaml*. Available at: <http://www.seas.upenn.edu/~hongboz/main.pdf>. Accessed 11/2013.
- [Huang05] Shan Shan Huang, David Zook, and Yannis Smaragdakis. 2005. *Statically safe program generation with SafeGen*. In Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE 2005), Tallinn, Estonia, September 29- October 1, Springer LNCS 3676, pp. 309-326. Available at: http://dx.doi.org/10.1007/11561347_21.
- [Huang08] Shan Shan Huang and Yannis Smaragdakis. 2008. *Expressive and safe static reflection with MorphJ*. SIGPLAN Not. 43, 6 (June 2008), pp. 79-89. Available at: <http://doi.acm.org/10.1145/1379022.1375592>.
- [Huang11] Shan Shan Huang and Yannis Smaragdakis. 2011. *Morphing: Structurally shaping a class by reflecting on others*. ACM Trans. Program. Lang. Syst. 33, 2, Article 6 (February 2011), 44 pages. Available at: <http://doi.acm.org/10.1145/1890028.1890029>.
- [Hurtado] José Antonio Hurtado. 2006. *wxFormBuilder: A RAD tool for wx GUIs*. <http://sourceforge.net/projects/wxformbuilder/> Accessed 11/2013.

- [Ierusalimschy] Roberto Ierusalimschy. 2013. *Programming in Lua, Third Edition*. Lua.org, ISBN-13: 978-8590379850.
- [JetBrains] JetBrains. 2013. *IntelliJIDEA - Groovy and Grails*. http://www.jetbrains.com/idea/features/groovy_grails.html. Accessed 11/2013.
- [Johnson] Rod Johnson. 2011. Aspect Oriented Programming with Spring. <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html>. Accessed 11/2013.
- [Jones] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. Available at: <http://www.itu.dk/~sestoft/pebook/jonesgomardsestoft-a4.pdf>. Accessed 11/2013.
- [Kaewkasi] Chanwit Kaewkasi and John R. Gurd. 2008. *Groovy AOP: a dynamic AOP system for a JVM-based language*. In Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies (SPLAT '08). ACM, New York, NY, USA, Article 3, 6 pages. Available at: <http://doi.acm.org/10.1145/1408647.1408650>.
- [Kameyama08] Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2008. *Closing the stage: from staged code to typed closures*. In Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM '08). ACM, New York, NY, USA, pp. 147-157. Available at: <http://doi.acm.org/10.1145/1328408.1328430>.
- [Kameyama09] Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2009. *Shifting the stage: staging with delimited control*. In Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '09). ACM, New York, NY, USA, pp. 111-120. Available at: <http://doi.acm.org/10.1145/1480945.1480962>.
- [Kamin] Sam Kamin, Lars Clausen and Ava Jarvis. 2003. *Jumbo: Run-time Code Generation for Java and its Applications*. In Proceedings of the 1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO '03), 23-26 March, San Francisco, California, USA, pp. 48-56. Available at: <http://doi.ieeecomputersociety.org/10.1109/CGO.2003.1191532>.
- [Kent] Stuart Kent. 2002. *Model Driven Engineering*. In Proceedings of the Third International Conference on Integrated Formal Methods, IFM2002, Turku, Finland, May 15-18, Springer LNCS 2335, pp. 286-298. Available at: http://dx.doi.org/10.1007/3-540-47884-1_16.
- [Kernighan] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C programming language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition.
- [Kiczales91] Gregor Kiczales, Jim Rivieres, and Daniel G. Bobrow. 1991. *The Art of the Metaobject Protocol*. The MIT Press. ISBN-13: 978-0262610742.
- [Kiczales97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. 1997. *Aspect-Oriented Programming*. In Proceedings of the 11th European

- Conference on Object-Oriented Programming (ECOOP '97), Jyväskylä, Finland, June 9-13, Springer LNCS 1241, pp. 220-242. Available at: <http://dx.doi.org/10.1007/BFb0053381>.
- [Kiczales01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. 2001. *An overview of AspectJ*. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01), Budapest, Hungary, June 18-22, Springer LNCS 2072, pp. 327-354. Available at: http://dx.doi.org/10.1007/3-540-45337-7_18.
- [Kohlbecker] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. *Hygienic macro expansion*. In Proceedings of the 1986 ACM conference on LISP and functional programming (LFP '86). ACM, New York, NY, USA, pp. 151-161. Available at: <http://doi.acm.org/10.1145/319838.319859>.
- [Kokaji] Yuichiro Kokaji and Yuki Yoshi Kameyama. 2011. *Polymorphic multi-stage language with control effects*. In Proceedings of the 9th Asian conference on Programming Languages and Systems (APLAS'11), Kenting, Taiwan, December 5-7, Springer LNCS 7078, pp. 105-120. Available at: http://dx.doi.org/10.1007/978-3-642-25318-8_11.
- [Laddad03] Ramnivas Laddad. 2003. *AspectJ in Action - Practical Aspect-Oriented Programming*. Manning. ISBN: 1930110936.
- [Laddad06] Ramnivas Laddad. 2006. *Aspect Oriented Refactoring*. Addison-Wesley Professional. ISBN: 978-0321304728.
- [Lee] Byeongcheol Lee, Robert Grimm, Martin Hirzel, and Kathryn S. McKinley. 2012. *Marco: safe, expressive macros for any language*. In Proceedings of the 26th European conference on Object-Oriented Programming (ECOOP'12), Beijing, China, June 11-16, Springer LNCS 7313, pp. 589-613. Available at: http://dx.doi.org/10.1007/978-3-642-31057-7_26.
- [Lerner] Barbara Staudt Lerner, Stefan Christov, Alexander Wise, and Leon J. Osterweil. 2008. *Exception handling patterns for processes*. In Proceedings of the 4th international workshop on Exception handling (WEH '08). ACM, New York, NY, USA, pp. 55-61. Available at: <http://doi.acm.org/10.1145/1454268.1454276>.
- [Lilis12a] Yannis Lilis and Antony Savidis. 2012. *Supporting Compile-Time Debugging and Precise Error Reporting in Meta-Programs*. In the 50th International Conference on Objects, Models, Components, Patterns (TOOLS 2012), 29-31 May, Prague, Czech Republic, Springer LNCS 7304, pp. 155-170. Available at: http://dx.doi.org/10.1007/978-3-642-30561-0_12.
- [Lilis12b] Yannis Lilis and Antony Savidis. 2012. *Implementing Reusable Exception Handling Patterns with Compile-Time Metaprogramming*. In Proceedings of the 4th International Workshop on Software Engineering for Resilient Systems (SERENE 2012), 27-28 September, Pisa, Italy, Springer LNCS 7527, pp. 1-15. Available at: http://dx.doi.org/10.1007/978-3-642-33176-3_1.

- [Lilis13] Yannis Lilis and Antony Savidis. *An Integrated Implementation Framework for Compile-Time Metaprogramming*. Software: Practice and Experience. To appear.
- [Lippert] Martin Lippert and Cristina Videira Lopes. 2000. *A study on exception detection and handling using aspect-oriented programming*. In Proceedings of the 22nd international conference on Software engineering (ICSE '00). ACM, New York, USA, Limerick, Ireland, pp. 418-427. Available at: <http://doi.acm.org/10.1145/337180.337229>.
- [LispWorks] LispWorks Ltd. 2012. *LispWorks 6.1*. <http://www.lispworks.com/>. Accessed 11/2013.
- [Mainland] Geoffrey Mainland. 2007. *Why it's nice to be quoted: quasiquoting for haskell*. In Proceedings of the ACM SIGPLAN workshop on Haskell workshop (Haskell '07). ACM, New York, NY, USA, pp. 73-82. Available at: <http://doi.acm.org/10.1145/1291201.1291211>.
- [McCarthy] John McCarthy. 1962. *LISP 1.5 Programmer's Manual*. MIT Press.
- [McCune] Tim McCune. 2006. *Exception-handling antipatterns*. <https://today.java.net/article/2006/04/04/exception-handling-antipatterns>. Accessed 11/2013.
- [Meyer91] Bertrand Meyer. 1991. Design by Contract. In *Advances in Object-Oriented Software Engineering*. Prentice Hall, pp. 1-50.
- [Meyer92] Bertrand Meyer. 1992. *Eiffel: the Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-247925-7.
- [Miao] Weiyu Miao and Jeremy Siek. 2012. *Pattern-based traits*. In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12). ACM, New York, NY, USA, pp. 1729-1736. Available at: <http://doi.acm.org/10.1145/2245276.2232057>.
- [Microsoft] Microsoft Corporation. *Using IntelliSense*. <http://msdn.microsoft.com/en-us/library/hcwl569b.aspx>. Accessed 11/ 2013.
- [Moore] J. Strother Moore. 1976. *The InterLisp Virtual Machine Specification*. Technical Report CSL 76-5, Xerox Palo Alto Research Center. Available at: <http://www.cs.utexas.edu/~moore/publications/interlisp-vm.pdf>. Accessed 11/2013.
- [Neverov04] Gregory Neverov and Paul Roe. 2004. *Metaphor: A Multi-Stage, Object-Oriented Programming Language*. In Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE '04), October 24-28, Vancouver, Canada, Springer LNCS 3286, pp. 168-185. Available at: http://dx.doi.org/10.1007/978-3-540-30175-2_9.
- [Neverov06] Gregory Neverov and Paul Roe. 2006. *Experiences with an object-oriented, multi-stage language*. Science of Computer Programming, Volume 62, Issue 1, September 2006, pp. 85-94, Available at: <http://dx.doi.org/10.1016/j.scico.2006.05.002>.

- [Newton] Keenan Newton. 2007. *The Exception Handling Application Block*. From *The Definitive Guide to the Microsoft Enterprise Library*, pp. 221-257. Apress.
- [Ng] Karen Ng, Matt Warren, Peter Golde and Anders Hejlsberg. 2012. *The Roslyn Project – Exposing the C# and VB compiler’s code analysis*. <http://go.microsoft.com/fwlink/?LinkID=230702>. Accessed 11/2013.
- [Nicklisch] Jürgen Nicklisch-Franken, Hamish Mackenzie, Andrew U. Frank and Christian Gruber. 2010. *An Integrated Development Environment for Haskell*. Available at: http://leksah.org/leksah_manual.pdf. Accessed 11/2013.
- [Nizhegorodov] Dmitry Nizhegorodov. 2000. *Jasper: Type-Safe MOP-Based Language Extensions and Reflective Template Processing in Java*. In *Proceedings of ECOOP’2000 Workshop on Reflection and Metalevel Architectures: State of the Art, and Future Trends*, ACM Press (2000). Available at: <ftp://ftp.disi.unige.it/person/CazzolaW/EWRMA/jasper-reflection.pdf>. Accessed 11/2013.
- [Odersky] Martin Odersky. 2013. *The Scala Language Specification. Version 2.8*. <http://www.scala-lang.org/files/archive/nightly/pdfs/ScalaReference.pdf>. Accessed 11/2013.
- [Oiwa] Yutaka Oiwa, Hidehiko Masuhara, and Akinori Yonezawa. 2001. *DynJava: Type safe dynamic code generation in Java*. In *Proceedings of the 3rd JSSST Workshop on Programming and Programming Languages (PPL ’01)*, Tokyo, Japan. Available at: <http://loome.cs.illinois.edu/CS498F10/readings/dynjava.pdf>. Accessed 11/2013.
- [Oracle] Oracle. *Java SE 7 Java Platform Debugger Architecture*. <http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/index.html>. Accessed 11/2013.
- [OMG10] Object Management Group. 2010. *OMG Model Driven Architecture - The Architecture of Choice for a Changing World*. <http://www.omg.org/mda/> Accessed 11/2013.
- [OMG12] Object Management Group. 2012. *Object Management Group Object Constraint Language (OCL)*. Available at: <http://www.omg.org/spec/OCL/ISO/19507/PDF/> Accessed 11/2013.
- [Palmer] Zachary Palmer and Scott F. Smith. 2011. *Backstage Java: making a difference in metaprogramming*. In *Proceedings of the 2011 ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA ’11)*. ACM, New York, NY, USA, pp. 939-958. Available at: <http://doi.acm.org/10.1145/2048066.2048137>.
- [Pawlak] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. 2001. *JAC: A Flexible Solution for Aspect-Oriented Programming in Java*. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION ’01)*, Kyoto, Japan, September 25–28,

- Springer LNCS 2192, pp 1-24. Available at: http://dx.doi.org/10.1007/3-540-45429-2_1.
- [Porkolab] Zoltan Porkolab, Jozsef Mihalicza and Adam Sipos. 2006. *Debugging C++ template metaprograms*. In Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06), ACM, New York, NY, USA, pp. 255-264. Available at: <http://dx.doi.org/10.1145/1173706.1173746>.
- [Rauglaudre] Daniel de Rauglaudre. 2003. *Camlp4 – Tutorial*. Available at: <http://caml.inria.fr/pub/docs/tutorial-camlp4/index.html>. Accessed 11/2013.
- [Reppy] John Reppy and Aaron Turon. 2007. Metaprogramming with Traits. In Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP 2007), Berlin, Germany, July 30 - August 3, Springer LNCS 4609, pp. 373-398. Available at: http://dx.doi.org/10.1007/978-3-540-73589-2_18.
- [Rhiger] Morten Rhiger. 2012. *Staged computation with staged lexical scope*. In Proceedings of the 21st European conference on Programming Languages and Systems (ESOP'12), Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, Springer LNCS 7211, pp. 559-578. Available at: http://dx.doi.org/10.1007/978-3-642-28869-2_28.
- [Riehl] Jonathan Riehl. 2009. *Language embedding and optimization in mython*. In Proceedings of the 5th symposium on Dynamic languages (DLS '09). ACM, New York, NY, USA, pp. 39-48. Available at: <http://doi.acm.org/10.1145/1640134.1640141>.
- [Rompf] Tiark Rompf and Martin Odersky. 2010. *Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs*. In Proceedings of the ninth international conference on Generative programming and component engineering (GPCE '10). ACM, New York, NY, USA, pp. 127-136. Available at: <http://doi.acm.org/10.1145/1868294.1868314>.
- [Rudolph] Johannes Rudolph and Peter Thiemann. 2010. *Mnemonics: type-safe bytecode generation at run time*. In Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '10). ACM, New York, NY, USA, pp. 15-24. Available at: <http://doi.acm.org/10.1145/1706356.1706361>.
- [Savidis05] Antony Savidis. 2005. *Dynamic Imperative Languages for Runtime Extensible Semantics and Polymorphic Meta-Programming*. In Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2005), 8-9 September, Heraklion, Crete, Greece, Springer LNCS 3943, pp. 113-128. Available at: http://dx.doi.org/10.1007/11751113_9.
- [Savidis07] Antony Savidis, Themistoklis Bourdenas and Yannis Georgalis. 2007. *An Adaptable Circular Meta-IDE for a Dynamic Programming Language*. In Proceedings of the 4th international workshop on Rapid Integration of Software Engineering Techniques (RISE 2007), 26-27

- November, Luxemburg pp. 99-114. Available at: <http://www.ics.forth.gr/hci/files/plang/sparrow.pdf>. Accessed 11/2013.
- [Savidis10] Antony Savidis. 2010. *The Delta Programming Language*. <http://www.ics.forth.gr/hci/files/plang/Delta/Delta.html>. Accessed 11/2013.
- [Schmidt] Douglas C. Schmidt. 2006. *Model-driven engineering*. Computer-IEEE Computer Society, 39(2), 25-31. Available at: <http://dx.doi.org/10.1109/MC.2006.58>.
- [Seibel] Peter Seibel. 2005. *Practical Common Lisp*. Apress, ISBN 978-1590592397.
- [Servetto10] Marco Servetto and Elena Zucca. 2010. *MetaFJig: a meta-circular composition language for Java-like classes*. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10). ACM, New York, NY, USA, pp. 464-483. Available at: <http://doi.acm.org/10.1145/1869459.1869498>.
- [Servetto13] Marco Servetto and Elena Zucca. 2013. *A meta-circular language for active libraries*. In Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation (PEPM '13). ACM, New York, NY, USA, pp. 117-126. Available at: <http://doi.acm.org/10.1145/2426890.2426913>.
- [Sheard98] Tim Sheard. 1998. *Using MetaML: A Staged Programming Language*. In: Advanced Functional Programming. 12-19 September, Braga, Portugal, Springer LNCS 1608, pp. 207-239. Available at: http://dx.doi.org/10.1007/10704973_5.
- [Sheard00] Tim Sheard, Zino Benaissa, and Matthieu Martel. 2000. *Introduction to Multistage Programming Using MetaML*. Pacific Software Research Center, Oregon Graduate Institute, 2nd edition. Available at: <http://web.cecs.pdx.edu/~sheard/papers/manual.ps>. Accessed 11/2013.
- [Sheard01] Tim Sheard. 2001. *Accomplishments and research challenges in metaprogramming*. In Proceedings of the Second International Workshop on Semantics, Application and Implementation of Program Generation (SAIG'01), 6 September, Florence, Italy, Springer LNCS 2196, pp. 2-44. Available at: http://dx.doi.org/10.1007/3-540-44806-3_2.
- [Sheard02] Tim Sheard and Simon Peyton Jones. 2002. *Template metaprogramming for Haskell*. SIGPLAN Not. 37, 12, December 2002, pp. 60-75. Available at: <http://dx.doi.org/10.1145/636517.636528>.
- [Skalski04] Kamil Skalski, Michal Moskal and Pawel Olszta. 2004. *Metaprogramming in Nemerle*. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.8265&rep=rep1&type=pdf>. Accessed 11/2013.
- [Skalski05] Kamil Skalski. 2005. *Syntax-extending and type-reflecting macros in an object-oriented language*. Master's Thesis. Available at: <http://nazgul.omega.pl/macros.pdf>. Accessed 11/2013.

- [Smith] Randall B. Smith and David Ungar. 1995. *Programming as an Experience: The Inspiration for Self*. In Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95), Aarhus, Denmark, Springer LNCS 952, pp. 303–330. Available at: http://dx.doi.org/10.1007/3-540-49538-X_15.
- [Subramaniam] Venkat Subramaniam. 2013. *Programming Groovy 2: Dynamic Productivity for the Java Developer*. Pragmatic Bookshelf.
- [Syme] Don Syme. 2006. *Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution*. In Proceedings of the 2006 workshop on ML (ML '06). ACM, New York, NY, USA, pp. 43-54. Available at: <http://doi.acm.org/10.1145/1159876.1159884>.
- [Steele] Guy L. Steele. 1990. *Common Lisp: The Language*. Digital Press, second edition, ISBN 1-55558-041-6.
- [Stroustrup] Bjarne Stroustrup. 2000. *The C++ Programming Language Special Edition*. Addison-Wesley.
- [Taha97] Walid Taha and Tim Sheard. 1997. *Multi-stage programming with explicit annotations*. In Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM '97), ACM, New York, NY, USA pp. 203-217, December 1997. Available at: <http://doi.acm.org/10.1145/258994.259019>.
- [Taha04] Walid Taha. 2004. *A gentle introduction to multi-stage programming*. In Domain-Specific Program Generation, Germany, March 2003, C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds. Springer LNCS 3016, 30-50. Available at: http://dx.doi.org/10.1007/978-3-540-25935-0_3.
- [Tatsubori] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. 2000. *OpenJava: A Class-Based Macro System for Java*. In Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering, Springer LNCS 1826, pp. 117-133. Available at: http://dx.doi.org/10.1007/3-540-45046-7_7.
- [Tratt05] Laurence Tratt. 2005. *Compile-time meta-programming in a dynamically typed OO language*. In Proceedings of the 2005 Symposium on Dynamic Languages (DLS '05). ACM, New York, USA, pp. 49-63. Available at: <http://doi.acm.org/10.1145/1146841.1146846>.
- [Tratt08] Laurence Tratt. 2008. *Domain specific language implementation via compile-time metaprogramming*. ACM Transactions on Programming Languages and Systems TOPLAS, 30(6), pp. 1-40, October 2008. Available at: <http://doi.acm.org/10.1145/1391956.1391958>.
- [Turner] Kenneth J. Turner. 1994. *Exploiting the m4 macro language*. Technical Report CSM-126, Department of Computing Science and Mathematics, University of Stirling, Scotland, September 1994. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.7662&rep=rep1&type=pdf>. Accessed 11/2013.

- [Valsamakis] Yannis Valsamakis. 2013. Improved Model-Driven Engineering with Staged Code Generators. Master's Thesis.
- [Veldhuizen96] Todd Veldhuizen. 1996. *Using C++ template metaprograms*. In C++ gems, Stanley B. Lippman (Ed.). Sigs Reference Library Series, Vol. 5. SIGS Publications, Inc., New York, NY, USA 459-473.
- [Veldhuizen03] Todd Veldhuizen. 2003. *C++ templates are Turing Complete*. Technical Report, Indiana University Computer Science. Available at: <http://ubietylab.net/ubigraph/content/Papers/pdf/CppTuring.pdf>. Accessed 11/2013.
- [Viera] Marcos Viera and Alberto Pardo. 2006. *A multi-stage language with intensional analysis*. In Proceedings of the 5th international conference on Generative programming and component engineering (GPCE '06). ACM, New York, NY, USA, pp. 11-20. Available at: <http://doi.acm.org/10.1145/1173706.1173709>.
- [Weise] Daniel Weise and Roger Crew. 1993. *Programmable syntax macros*. In Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93), Robert Cartwright (Ed.). ACM, New York, NY, USA, pp. 156-165, June 1993. Available at: <http://doi.acm.org/10.1145/155090.155105>.
- [Westbrook] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. 2010. *Mint: Java multi-stage programming using weak separability*. In Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI '10). ACM, New York, NY, USA, pp. 400-411. Available at: <http://doi.acm.org/10.1145/1806596.1806642>.
- [WirfsBrock] Rebecca Wirfs-Brock. 2002. *What it Really Takes to Handle Exceptions*. In: forUse 2002 Conference Proceedings. pp. 341-370. Available at: <http://www.foruse.com/articles/exceptions.pdf>. Accessed 11/2013.
- [Yao] Zhen Yao, Qi-long Zheng, and Guo-liang Chen. 2005. *AOP++: a generic aspect-oriented programming framework in C++*. In Proceedings of the 4th international conference on Generative Programming and Component Engineering (GPCE'05), Tallinn, Estonia, September 29 - October 1, Springer LNCS 3676, pp. 94-108. Available at: http://dx.doi.org/10.1007/11561347_8.
- [Yin] Haihan Yin, Christoph Bockisch, and Mehmet Aksit. 2012. *A fine-grained debugger for aspect-oriented programming*. In Proceedings of the 11th annual international conference on Aspect-oriented Software Development (AOSD '12). ACM, New York, NY, USA, pp. 59-70. Available at: <http://doi.acm.org/10.1145/2162049.2162057>.
- [Zook] David Zook, Shan Shan Huang and Yannis Smaragdakis. 2004. *Generating AspectJ Programs with Meta-AspectJ*. In Proceedings of the Third International Conference on Generative Programming and Component Engineering, GPCE 2004, Vancouver, Canada, October 24-28, pp. 1-18, Available at: http://dx.doi.org/10.1007/978-3-540-30175-2_1.