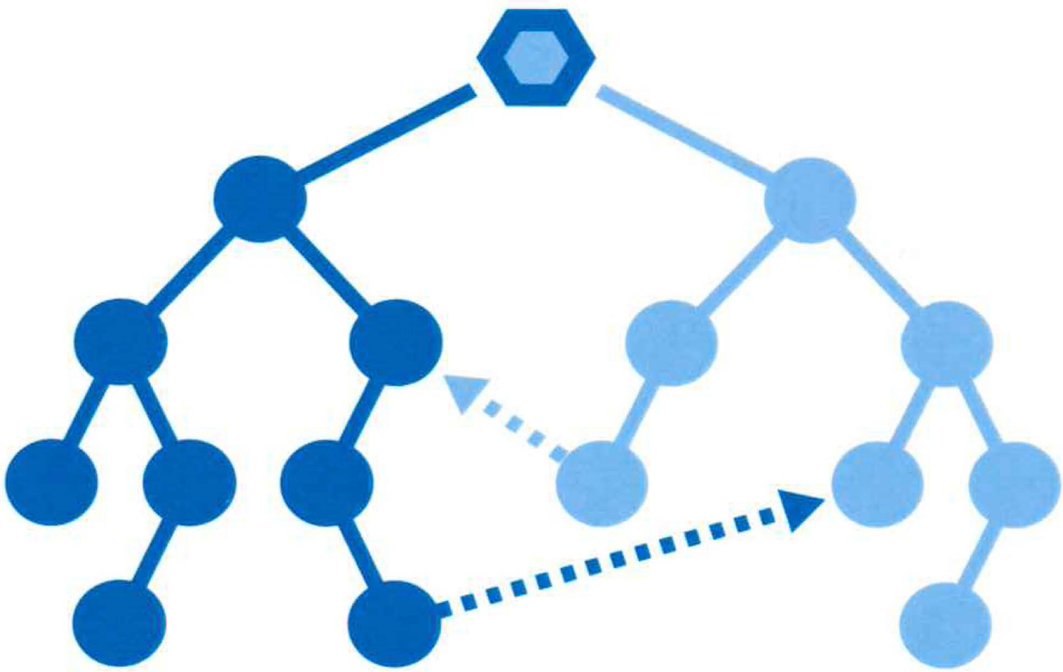
SE | RWTH AACHEN UNIVERSITY

Pedram Mir Seyed Nazari

MontiCore: Efficient Development of Composed Modeling Language Essentials



Aachener Informatik-Berichte,
Software Engineering

Band 29

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

MontiCore: Efficient Development of Composed Modeling Language Essentials

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Pedram Mir Seyed Nazari
aus Teheran, Iran

Berichter: Universitätsprofessor Dr. Bernhard Rumpe
Universitätsprofessor Dr. Uwe Aßmann

Tag der mündlichen Prüfung: 6. Februar 2017

[Naz17] P. Mir Seyed Nazari:
MontiCore: Efficient Development of Composed Modeling Language Essentials.
Shaker Verlag, ISBN 978-3-8440-5320-3. Aachener Informatik-Berichte, Software Engineering, Band 29. 2017.
www.se-rwth.de/publications/

Abstract

Model-driven engineering exploits models as first-class artifacts to tackle the complexity and heterogeneity of large software systems. Such models can be built with domain-specific languages (DSLs) that enable higher-levels of abstraction and also facilitate separation of concerns and reuse. Unlike general-purpose languages, a DSL allows specification of system aspects by using the terminology of the domain. This, in turn, enables domain experts—who rarely have software engineering skills—to become directly involved in the development process.

In textual, grammar-based languages, an abstract syntax tree (AST) technically represents the model within a tool and thus serves as the central artifact for model processing, e.g., for static analysis or code generation. Deriving the AST directly from the grammar keeps it consistent with the concrete syntax [KRV07b] and, hence, reduces maintenance efforts. Thus, the AST greatly depends on the grammar in terms of both content and structure, leading to two drawbacks: First, since the AST does not necessarily provide a model's essential information (e.g., the element a name refers to) in a straightforward manner, this complexity can hamper the development of tools processing that model. Second, even small grammar changes can affect the AST that, in turn, may require dependent tools to be updated.

Moreover, since heterogeneous languages are typically required to specify different aspects of a software system, another problem is that models of those languages first have to be integrated before they can be analyzed and synthesized together [DBC+15]. Therefore, each model has to have an interface to enable composition with other models [HR13]. The composition can include models defined within the same language as well as models of independent, heterogeneous languages. Again, the AST does not explicitly exhibit a model's interface but instead mixes it with other, unessential information.

To address the above issues, this dissertation aims to promote the development of an additional structure (called ST) which captures (i) information that is essential for processing models of a language as well as (ii) a language's interface to enable composition of models of both the same language and heterogeneous languages. Unlike the (generated) AST, the ST can also contain information that is not directly defined in the model but related to it (e.g., all states that are reachable from a specific state of an automaton). Tools can employ the ST together with the AST to access the relevant information as needed, which facilitates model processing.

An additional structure, however, requires development effort itself. To further an efficient and effective development of STs, this dissertation presents a generic infrastructure with reasonable defaults. Moreover, the infrastructure provides generic concepts and mechanisms for defining model interfaces (as part of the respective STs) to allow an efficient composition of heterogeneous models via their interfaces in a non-invasive way. Thus, the models can be reused in various contexts. Furthermore, based on the

generic infrastructure, methods and patterns for developing language-specific STs are elaborated. To reduce the amount of handcrafted boilerplate code as well as the number of programming errors, a generative approach is employed, which produces parts of the language-specific ST infrastructure and provides ways for efficiently extending and customizing it. The generated ST targets essential elements of a language which are unlikely to change.

Ultimately, the proposed ST approach (including its concepts and methods) simplifies the development of tools for model processing by explicitly providing the essential information of a model. In addition, the maintenance of such tools will be improved, since, compared to the AST, the ST is more robust against changes in the grammar. Finally, the proposed ST approach provides model interfaces more explicitly, thereby facilitating the composition of models even from heterogeneous languages.

Kurzfassung

In der Modell-getriebenen Softwareentwicklung werden Modelle als Kernartefakte verwendet, um die Komplexität und Heterogenität von großen Softwaresystemen zu beherrschen. Dabei können die Modelle mittels Domänen-spezifischer Sprachen (DSLs) erstellt werden, welche eine höhere Abstraktion ermöglichen und außerdem die Trennung unterschiedlicher Aspekte und deren Wiederverwendung erleichtern. Anders als Allzweck-Programmiersprachen, können die Systemaspekte mithilfe einer DSL in der Terminologie der jeweiligen Domänen spezifiziert werden. Auf diese Weise können Domänenexperten, welche selten Erfahrungen in Softwareentwicklung haben, direkt in den Entwicklungsprozess involviert werden.

In textuellen, Grammatik-basierten Sprachen wird ein Modell technisch durch einen abstrakten Syntaxbaum (AST) innerhalb eines Werkzeugs dargestellt. Der AST dient dabei als zentrales Artefakt für die Modellverarbeitung, zum Beispiel für statische Analysen oder für die Code-Generierung. Wird der AST direkt aus der Grammatik generiert, bleibt er konsistent mit der konkreten Syntax [KRV07b] und reduziert so den Wartungsaufwand. Als Konsequenz ist der AST sowohl inhaltlich als auch strukturell stark von der Grammatik abhängig, was zwei Nachteile mit sich bringt: Zum einen liefert der AST die essentiellen Informationen eines Modells (z.B. das Element, welches durch einen Namen referenziert wird) nicht notwendigerweise in einer gut aufbereiteten Form, was die Entwicklung von Werkzeugen zur Verarbeitung des Modells erschweren kann. Zum anderen können sogar kleinere Änderungen an der Grammatik den AST betreffen, was wiederum eine Anpassung der davon abhängigen Werkzeuge erfordern kann.

Um die verschiedenen Aspekte eines Softwaresystems zu spezifizieren, werden typischerweise heterogene Sprachen benötigt. Dabei müssen die Modelle dieser Sprachen erst integriert werden bevor sie gemeinsam analysiert und synthetisiert werden können [DBC+15]. Dazu muss jedes Modell eine Schnittstelle zur Verfügung stellen, um eine Komposition mit anderen Modellen zu ermöglichen [HR13]. Eine solche Komposition kann sowohl Modelle derselben Sprache als auch Modelle aus unabhängigen, heterogenen Sprachen beinhalten. Der AST ist dafür nur teilweise geeignet, da er die Schnittstelle eines Modells nicht explizit zur Verfügung stellt, sondern diese mit anderen, unwesentlichen Informationen vermischt.

Das Ziel dieser Dissertation ist es die Entwicklung einer zusätzlichen Struktur (ST) voranzutreiben, welche (i) Informationen erfasst, die essentiell für die Verarbeitung von Modellen einer Sprache sind und (ii) die Schnittstelle einer Sprache zur Verfügung stellt, um eine Komposition von Modellen aus derselben als auch aus heterogenen Sprachen zu ermöglichen. Im Gegensatz zum (generierten) AST kann die ST auch Informationen enthalten, die nicht direkt im Modell definiert diesem aber zugehörig sind (z.B. alle erreichbaren Zustände, ausgehend von einem bestimmten Zustand eines Automaten). Um die Verarbeitung von Modellen und somit den Zugriff auf die benötigten relevanten

Informationen zu vereinfachen, kann die ST zusammen mit dem AST von Werkzeugen verwendet werden.

Des Weiteren präsentiert diese Dissertation eine generische Infrastruktur mit umfangreichen Standardimplementierungen, um die effiziente und effektive Entwicklung von STs zu unterstützen. Die Infrastruktur stellt generische Konzepte und Mechanismen für die Erstellung von Modellschnittstellen (als Teil der zugehörigen STs) zur Verfügung, um eine effiziente, nicht-invasive Komposition von heterogenen Modellen über diese Schnittstellen zu ermöglichen. Somit können die Modelle in verschiedenen Kontexten wiederverwendet werden. Darüber hinaus werden basierend auf der generischen Infrastruktur Methodiken und Entwurfsmuster für die Entwicklung von sprachspezifischen STs ausgearbeitet. Um die Menge von handgeschriebenem Boilerplate Code und auch die Anzahl von Programmierfehlern zu reduzieren, wird ein generativer Ansatz angewendet, welcher Teile einer sprachspezifischen ST generiert und Möglichkeiten bietet diese Teile effizient zu erweitern und anzupassen. Die generierte ST fokussiert die essentiellen Elemente einer Sprache, welche selten verändert werden.

Zusammengefasst vereinfacht der in dieser Arbeit vorgestellte Ansatz (mit den Konzepten und Methodiken) die Entwicklung von Werkzeugen zur Modellverarbeitung, indem die ST die essentiellen Informationen eines Modells explizit zur Verfügung stellt. Des Weiteren wird die Wartung solcher Werkzeuge verbessert, da die ST (im Vergleich zum AST) robuster im Hinblick auf Grammatikänderungen ist. Zusätzlich wird die Komposition von Modellen aus heterogenen Sprachen vereinfacht, da die ST Modellschnittstellen expliziter zur Verfügung stellt.

Danksagung

An dieser Stelle möchte ich mich bei den lieben Menschen bedanken, die mich während meiner Promotion unterstützt und so zum Erfolg dieser Dissertation beigetragen haben.

Mein ganz besonderer Dank gilt meinem Doktorvater Prof. Dr. Bernhard Rumpe für die spannende und lehrreiche Zeit am Lehrstuhl, für die ideenreichen und konstruktiven Diskussionen, für das in mich gesetzte Vertrauen zur Leitung der Modellierungsgruppe und für die Möglichkeit an zahlreichen herausfordernden Industrie- und Forschungsprojekten mitarbeiten zu können. Mit seinen Ratschlägen und weitsichtigen Anregungen hat Bernhard maßgeblich zum Erfolg dieser Arbeit beigetragen.

Prof. Dr. Uwe Aßmann danke ich herzlich für sein Interesse an meiner Arbeit und die Übernahme des Zweitgutachtens. Ich möchte mich ebenfalls bei Prof. Dr. Ulrik Schroeder für die Leitung meines Promotionskomitees und bei Prof. Dr. Erika Ábrahám für die Durchführung der Theorieprüfung bedanken. Darüber hinaus möchte ich mich herzlich bei Prof. Dr. Manfred Nagl für die zahlreichen interessanten Gespräche und hilfreichen Denkanstöße bedanken. Sein breites und detailreiches Wissen weit über die Informatik hinaus wird mich immer faszinieren.

Dr. Martin Schindler danke ich sehr für das anfängliche Mentoring sowie für seine fachliche und freundschaftliche Unterstützung während meiner Promotion. Ebenfalls danke ich Antonio "Toni" Navarro Pérez, der mit seiner herausragenden fachlichen Kompetenz zur Findung meines Forschungsthemas beigetragen hat. Sehr gerne denke ich an unsere philosophischen Gedankenexperimente zurück, die oft endeten wie sie anfingen: mit einem lachenden "Why?". Bei Michael von Wenckstern bedanke ich mich für unsere einzigartige Zeit als "digital Transformers" sowie für die zahlreichen und sehr spannenden Diskussionen über die unterschiedlichsten Themen. Alexander "Alex" Roth danke ich für die vielen Gespräche frühmorgens noch bevor der Lehrstuhl-Alltag begonnen hatte und für die gute Zusammenarbeit in Lehre und Forschung. Die Paper-Marathons gehören zu den Highlights meiner Zeit am Lehrstuhl. Dimitri Plotnikov bin ich dankbar für sein kontinuierliches Feedback zu MontiCore und der Symboltabelle. Danke auch an Klaus Müller für die spannenden Tischtennis Matches und die gute Zeit auf der Konferenz in Frankreich zusammen mit Alex. Vielen Dank an Deni Raco, der mithilfe einer Vielzahl von Kicker-, Poker- und Schachturnieren stets für einen guten Ausgleich gesorgt hat.

Darüber hinaus möchte ich mich ganz herzlich beim MontiCore Team bedanken, angefangen bei Robert Heim, mit dem ich zusammen die Planung und Analyse für das MontiCore Reengineering durchgeführt habe. Robert gehört zu den strukturiertesten und talentiertesten Menschen, die ich kennengelernt habe. Es war mir stets eine große Freude mit ihm zusammenzuarbeiten. Auch danke ich Katrin Hölldobler für die großartige Zusammenarbeit und ihr gutes Auge für Details. Andreas Horst hat leidenschaftlich dafür gesorgt, dass wir stets eine stabile und aktuelle Build-Infrastruktur hatten. Vielen Dank dafür. Des Weiteren danke ich Sebastian Oberhoff für die gute Unterstützung im

MontiCore Team. Ein sehr großer Dank gebührt Marita Breuer und Galina Volkova, die es mit ihrer Hilfsbereitschaft und der Übernahme diversester Aufgaben ermöglicht haben, MontiCore zum geplanten Zeitpunkt und in guter Qualität zu releasen. Ich danke ebenfalls den SE-Runners Michael, Alex und unserem Halbmarathonläufer Evgeny Kusmenko für die sportliche Abwechslung nach langen Diss-Tagen und das anschließende Entspannen bei einem guten Essen. Außerdem danke ich Kai Adam, Vincent Bertram, Lennart Bucher, Arvid Butting, Robert Eikermann, Timo Greifenberg, Dr. Arne Haber, Lars Hermerschmidt, Dr. Christoph Herrmann, Steffi Kaiser, Oliver Kautz, Carsten Kolassa, Thomas Kurpick, Achim Lindt, Dr. Markus Look, Matthias Markthaler, Dr. Cem Mengi, Dr. Claas Pinkernell, Manuel Pützer, Dr. Dirk Reiß, Dr. Jan Oliver Ringert, Christoph Schulze, Philipp Schütze, Brian Sinkovec, Max Voß und Dr. Andreas Wortmann für das exzellente Arbeitsklima und die spannende Zeit am Lehrstuhl. Ein großer Dank gebührt ebenfalls den zahlreichen Studenten, die durch ihre Abschlussarbeiten oder ihre Tätigkeiten als studentische Hilfskräfte zur Weiterentwicklung und Evaluation meiner Forschungsthemen beigetragen haben. Dazu gehören unter anderem: Odgerel Boldbaatar, Abdullah Celik, Peter Damm, Martin Hackenberg, Christoph Hommelsheim, Michael Krein, Markus Lüger, Mirko Mades, David Mularski, Stefan Nessel, Bao-Loc "Fe" Nguyen Ngo, Patrick Schlesiona, Simon Siewert und Alex Tritthart. Auch möchte ich Sylvia Gunder und Gabriele Heuschen danken, dass sie für einen angenehmen und reibungslosen Lehrstuhlbetrieb gesorgt haben. Vielen Dank an Robert, Katrin, Alex, Timo, Deni und Michael für das Korrekturlesen früher Fassungen der Dissertation.

Schließlich gilt mein unendlicher Dank meiner Familie. Meinen Eltern, Soudabeh und Mahmoud Nazari, danke ich von ganzem Herzen, dass Sie mich in jeder Situation meines Lebens unterstützt und immer alles daran gesetzt haben, dass ich meine Träume verwirklichen kann. Ohne Euch wäre diese Arbeit nicht möglich gewesen. Ich hoffe, dass ich Euch hiermit etwas zurückgeben konnte. Meinen Geschwistern Misam, Pegah und Navid Nazari danke ich für ihre endlose Unterstützung auf meinem Lebensweg und für die vielen schönen gemeinsamen Erlebnisse. Meiner süßen Nichte Nika danke ich, dass sie uns ohne Worte und mit nur einem Lächeln so viel Glück und Freude bereitet und unser Leben so sehr bereichert. Darüber hinaus danke ich meinen beiden Onkeln Siawash und Siamak Khajehi-Mahabadi, dass sie mir immer mit ihrer Erfahrung mit Rat und Tat zur Seite standen. Ich danke meinen Schwiegereltern Fereba und Nazier Chaled für ihre liebevolle Fürsorge und dass sie mich immer wie einen Sohn behandelt und an mich geglaubt haben. Meinen Schwägerinnen Shohreh Talebi und Natascha Chaled danke ich für ihre Unterstützung und die vielen tollen gemeinsamen Gespräche. Schließlich gebührt mein ganz besonderer Dank meiner wundervollen Frau Nilofar Nazari. Mit ihrer uneingeschränkten Unterstützung und Liebe und ihrer unglaublichen Geduld hat sie mich stets gestärkt und dadurch maßgeblich zum Gelingen dieser Arbeit beigetragen.

Aachen, April 2017
Pedram M. S. Nazari

Contents

Chapter 1

Introduction

Modern software systems are becoming increasingly complex and are pervasive in many domains, such as energy management, intelligent transportation, smart homes, and logistics (e.g., [FR07, Com15]). Prominent terms for complex software and software-intensive systems are *cloud-based systems*, *cyber-physical systems* (CPS) [Lee08] and *internet-of-things* (IoT) [AIM10]. They demand for distributed systems consisting of heterogeneous components both physical and logical. Moreover, many stakeholders (e.g., domain experts, application developers, user interface designers, etc.) are involved in the development process of such systems and address issues concerning different phases ranging from design, implementation to deployment [CCF+15].

Manually managing the complexity and heterogeneity of such systems with general-purpose languages (GPLs) is onerous and cost-intensive [FR07]. It further requires solutions that are hardly reusable in different contexts and are limited to specific platforms [KT08]. Model-driven engineering (MDE) [Sch06] tackles the complexity and heterogeneity of large software systems via dedicated *platform-independent* models, which describe various aspects of the system and that way enable separation of concerns. MDE tools can process those models and transform them to *platform-specific* target code. This approach not only facilitates reuse of the models for different contexts and platforms but also enables each stakeholder to focus on a particular aspect of the system [CCF+15].

To efficiently and effectively process a model, essential information related to it must be explicitly provided. Furthermore, several (heterogeneous) languages are required to specify different aspects of a software system [VWH09]. Hence, models of those languages must be integrated in order to be analyzed and synthesized together [DBC+15]. Therefore, each model has to provide an explicit interface to enable composition with other models [HR13]. The composition includes models defined within the same modeling language (e.g., class diagrams) as well as models of independent, heterogeneous modeling languages (e.g., class diagrams and statecharts). The latter in particular requires mappings between the models' interfaces [Rum13]. Modeling languages can be composed themselves, e.g., to enable model parts of a language to be embedded in models of other languages. For this, each language must exhibit an interface. In this respect, language composition can be considered as a special kind of model composition [Völ11].

Hence, the aim of this thesis is to support the language engineer (cf. Section 3.8) in

efficiently developing an infrastructure for capturing (i) information she deems essential for processing models of the language as well as (ii) model interfaces (determined by the modeling language's interface) to enable composition of models of both the same modeling language and heterogeneous modeling languages.

The remainder of this chapter is structured as follows. Section 1.1 introduces the context of this thesis and in particular previous work on which this thesis builds on. Further related work is discussed throughout the respective chapters. Section 1.2 states the research question of this thesis and outlines the thesis' main contributions to it. Finally, Section 1.3 outlines the structure of the current thesis.

1.1 Context of the Thesis

Language workbenches facilitate the development of modeling languages [Fow10]. The language workbench MontiCore [GKR+06, GKR+08, KRV08b, Kra10, KRV10] has been developed at the chair of Software Engineering at the RWTH Aachen university. It allows for an efficient and effective development of grammar-based, textual modeling languages. In MontiCore the abstract syntax tree (AST)[1] of a language is automatically derived from the grammar in order to keep it consistent with the concrete syntax [KRV07b]. That way, the AST holds the information (directly) defined in a model and can be employed, e.g., for code generation.

The AST, however, yields two major limitations. First, the AST highly depends on the grammar in terms of both content and structure (cf. [Fow10]). Therefore, it does not necessarily provide a model's (essential) information including its interface in a convenient way. Moreover, resulting from its tree-like structure, it does not allow for references between nodes, e.g., from different subtrees. Therefore, approaches such as [HM03, VS10, Bet13] offer ways for direct references between AST nodes[2]. Second, even small grammar changes can affect the AST that, in turn, may require dependent tools to be updated.

In order to tackle the above described limitations of an AST and at the same time provide relevant model information, Völkel employs so-called symbol tables (STs) [Völ11], where a symbol table consists of *generic* namespaces and *language-specific* symbol table entries (STEs)[3]. A STE represents a model element and its associated information.

[1] The AST is a tree representation of a parsed model and omits irrelevant syntactical information, such as semicolons or opening and closing brackets of a block.

[2] Older MontiCore versions allowed for simple references between AST nodes [KRV07b]. This feature has been removed in the current MontiCore version 4 since it, among others, hampers composition of models of different languages.

[3] Please note that the term "symbol table" used in [Völ11] goes beyond the classical definition where a symbol table allows "to find the record for each name quickly and to store or retrieve data from that record quickly" [ALSU06]. In Section 3.8 we define the term as understood in this thesis (and in MontiCore).

The current thesis builds on lessons learned from the work done in [Völ11]. We follow Völkel's approach to introduce an additional structure (i.e., the ST) besides the AST for the following reasons:

- An additional structure can represent the language's essence. i.e., its essential model. This includes essential information of a model such as its interface, which allows for encapsulating unnecessary information following Parnas' *information hiding* principle [Par71, Par72]. Fowler suggests to always create an essential model (which he calls semantic model[4]), among others, for the following reasons (cf. [Fow10]):
  - While the AST corresponds to the grammars structure, the essential model can differ substantially from it (as described above). In particular, the essential model is rarely a tree.
  - The essential model can be employed to test a model's semantics independently from the parsing process. In particular, different languages with different syntaxes can yield the same essential model.
  - Basing the code generation on the essential model, decouples it from the (abstract) syntax tree, and hence, from the grammar. Consequently, grammar evolution does not impact the code generation process as long as the essential model remains unchanged.
- The ST can contain information that is not directly defined in the model but somehow related to it. For example, a ST for a Java class can contain all direct and indirect supertypes of that class. Another example is a ST for a state of a statechart. That ST can provide information about all reachable states of the state. This especially facilitates the processing of the model since all relevant information is available in a convenient way.

Following a component-based software engineering approach [McI68], Völkel [Völ11] extended MontiCore with an infrastructure for a compositional development of a modeling language's concrete syntax, abstract syntax, symbol table[5], and context conditions. While his approach enables a declarative, generative approach for the concrete syntax as well as the abstract syntax, it provides a generic infrastructure for ST development with only little default implementations. However, many languages developed with this infrastructure (e.g., [HRR+11, HRRS11, HKR+11, Sch12, HRR12, Nes13, RRW13b, NPR13, HMSNR+15a, HHRW15, MSNR15a]) share many commonalities concerning their symbol tables, which leads to much repetitive code among these languages. The following gives an overview of major drawbacks and limitations regarding symbol tables in [Völ11]:

[4] We omit the term "semantic model" to avoid confusion with "semantics" as defined in [HR04].

[5] In the current thesis, we consider the symbol table to be part of the abstract syntax. Section 3.8 elaborates on this.

Missing link between AST and ST: AST and ST both offer important information about the processed models. Hence, depending on the task one or both structures are required. Therefore, they should be linked appropriately in order to allow access to either one in an easy and efficient way. In [Völ11] this link is only given to a certain extent.

Handcrafted boilerplate code: STs of different languages share many commonalities, such as how names are resolved and how models are loaded. The infrastructure provides reasonable defaults and (explicit) concepts for those commonalities only to some extent. As a result, much *handcrafted* boilerplate code exists among the tools developed with Völkel's infrastructure, which all require testing and maintenance. Providing default implementations as part of the generic infrastructure requires them to be tested and maintained only once, which improves the quality of the ST and increases its development efficiency.

Lack of methods and patterns: Furthermore, [Völ11] lacks comprehensive methods and patterns for developing language-specific symbol tables. This, however, is essential since implementing a ST requires a deep understanding of the underlying concepts. Moreover, the implementation of a language's ST can impact its composability with STs of other languages.

Inconsistency issues: Some model elements span a namespace. A Java class, for example, defines a namespace in which methods and fields can be defined. In [Völ11] the STE and its spanned namespace are not linked. In particular, there exists no explicit concept for a STE that spans a namespace. As a consequence, some information must be stored redundantly in both. This, however, can lead to inconsistencies between the STE and its spanned namespace which gives rise to different behavior depending on which one is used.

Focus on generic instead of specific aspects: Name-based model composition requires an underlying name resolution mechanism that finds the corresponding model element. The infrastructure in [Völ11] enables name resolution only via the *generic* namespaces, not via the *language-specific* STEs, emerging from the missing concept of a STE that spans a namespace mentioned in the previous item. This yields the drawback that the user of a ST (e.g., a code generator developer) must be aware of the underlying generic mechanism of the infrastructure, instead of focusing on specific aspects concerning the concrete language. This, among others, complicates usage and is more error-prone if used in an unintended manner. Furthermore, the resolution mechanism enables to compose models of heterogeneous languages by conducting translations between their elements. Therefore, only a generic usage of the ST guarantees the functioning of language composition, although the specific STEs provide the information in a more convenient way.

Inconvenient usage: Some important features of the ST can only be employed with additional infrastructure provided by MontiCore. For example, a name resolution cannot be started from a given namespace alone. Hence, it is not possible—at least not without helper classes—to pass (parts of) the ST as a self-contained construct to be used, e.g., for code generation. Instead, the required infrastructure must be available as well.

1.2 Objectives and Main Results

The main research question of this thesis is:

> *How can a generative approach support the efficient development of infrastructures for explicitly providing essential model information and that way facilitate processing of composed models from different, heterogeneous languages?*

The *essential information* of a model is the same independent of its (syntactical) representation (cf. [Bro87]) and is determined by the language engineer [HMSNR15b]. In particular, we aim at increasing *development efficiency* by reducing tedious work that the language engineer has to conduct manually. For this, we are interested in *generated* infrastructures to, among others, reduce the amount of handcrafted boilerplate code, reduce the error-proneness and that way increase a language's quality (cf. [VSB+13]). Following [SV06, KT08] we also aim at reducing the amount of generated code. Ideally, most of the code is part of a *generic framework* which not only "keeps the generator simpler" [KT08] but also increases the understandability and maintainability of the software. In MDE, *heterogeneous languages* are needed to specify the different concerns of a software system (e.g., [VWH09, CDB+14]). However, *composing models* of such languages requires additional effort, e.g., adaptation of their interfaces (cf. [Aßm03]).

The main contributions of this thesis are as follows:

- It provides a generic infrastructure called SMI (symbol management infrastructure) for efficient and effective development of modeling language symbol tables. SMI is integrated into the language workbench MontiCore. This thesis employs the idea of STs as in [Völ11] for realizing essential model information including their interfaces [HR13]. SMI provides reasonable defaults for concepts and mechanisms of STs that occur in many—block structured and lexically scoped (cf. Chapter 3)—modeling languages, e.g., [HRR12, Sch12, HMSNR+15a]. That way, it liberates the ST engineer (i.e., the language engineer) to a large extent from handcrafting boilerplate code.
- SMI is lightweight in the sense that it allows for developing self-contained STs, which can be used in a functional manner and without further technical infrastructure.

- SMI facilitates linking of AST and ST in both directions so that a ST user can easily choose either one as needed [HMSNR15b].
- SMI provides a generic name resolution mechanism, which can be customized for language-specific concerns. This mechanism enables name-based composition of models based on their interfaces (embodied in the STs). Composition of heterogeneous models additionally requires composition of the respective heterogeneous languages in order to make the languages' interfaces fit each other (i.e., adaptation [Aßm03]). The composition can be conducted in a non-invasive way [HLMSN+15a, HLMSN+15b, HMSNRW16], i.e., the involved languages do not have to be modified. Instead, glue code is needed (cf. [Aßm03]). This thesis focuses on the composition of STs (but not the concrete syntax or the AST of the models. For more information on these, please refer to [Völ11]).
- Several patterns for implementing language-specific STs with SMI are introduced and their impacts on language composition are discussed.
- Moreover, this thesis presents concepts and methods that allow for keeping generic (i.e., in SMI) and language-specific (i.e., handcrafted or generated) parts of the ST consistent by avoiding redundancy between them. This enables ST engineers to encapsulate the generic infrastructure so that ST users can focus on language-specific aspects while preserving functioning of language composition. Moreover, encapsulating the generic infrastructure liberates the user from understanding the underlying technical concepts and mechanisms.
- Although SMI reduces the amount of handcrafted code by providing reasonable defaults, it cannot completely eliminate boilerplate code. For example, language-specific classes still need to be manually created and integrated into the framework. In this thesis, we tackle this via a generative approach, which only led to a minimal extension of the MontiCore grammar. This does not only reduce the amount of handcrafted boilerplate code but also reduces the number of programming errors and increases the conformance of generated code to coding standards [Rum12].
- Certainly, a full-fledged ST generation is not possible (at least for more complex languages) since a language's ST infrastructure highly depends on the language engineer's design decisions [HMSNR15b]. Therefore, this thesis presents some approaches for efficiently extending and customizing the generated code based on [GHK+15a, GHK+15b] in order to meet language-specific requirements. Furthermore, the generated infrastructure can serve as starting point for the ST engineer since it, among others, demonstrates how language-specific parts are integrated into the generic infrastructure. Since the ST is highly language-specific, we do not introduce a new complex modeling language but instead allow the ST engineer to customize and extend the (generated) ST via the GPL Java [HMSNR15b].

Currently, SMI is already utilized in the following works (not necessarily by the author of this thesis):

- NESTML, a modeling language family for spiking neurons [PBI+16]
- MontiArc, an architecture description language [HRR12] (migrated from older MontiCore versions)
- Java 1.5 for MontiCore [Mul15]
- MontiJava, an extension of Java 1.5 [Mul15]
- Object Constraint Language (OCL) for MontiCore [Cel15]
- CD4Analysis, restricted UML/P class diagrams [Sch12, Rum16]
- UML/P object diagrams [Sch12, Rum16]
- JavaScript for MontiCore [Sie15]
- language family for robotics applications [HMSNR+15a] (migrated from older MontiCore versions)
- MontiCore grammar language [Kra10] (migrated from older MontiCore versions)
- managing guided and unguided code generator customizations [MSNRR15]
- managing the composition of output-specific generator information [MSNRR16]
- UML activity diagrams [LN16] based on [Rei16]
- tagging language for component and connector models [MRRW16]

Many of the above listed projects have been developed simultaneously with SMI which enabled to obtain fast feedback and improve SMI in the sense of its understandability for the language engineers and users, its applicability, and its development effort.

1.3 Structure of the Thesis

This thesis is structured as follows:

Chapter 1 gives an overview of the motivation, the context, and the goals of this thesis.

Chapter 2 describes terms of the model-driven engineering approach and introduces the MontiCore language workbench, which has been extended with the infrastructure developed in this thesis.

Chapter 3 elucidates core concepts and elements of a symbol table (as understood in the current thesis) and gives for each a clear definition. Furthermore, it describes the role of the symbol table as (part of) a modeling language's interface.

Chapter 4 presents the technical infrastructure developed during this thesis. The infrastructure provides generic classes for the concepts introduced in Chapter 3. Furthermore, Chapter 4 discusses several patterns for employing those generic classes for language-specific symbol tables. Moreover, it gives a naming convention for language-specific classes.

Chapter 5 presents a method as well as technical classes for building up a symbol table from a given AST so that the ST can be utilized, e.g., for name resolution (see next chapter). For this, it employs the technical classes introduced in Chapter 4.

Chapter 6 elaborates the general name resolution process developed in the current thesis and its respective technical realization (based on Chapter 4). The resolution is conducted on the symbol table (created as described in Chapter 5) and resolves symbols, i.e., model elements, among others, via their names. That way, it enables name-based model composition.

Chapter 7 applies some of the methods and technical classes introduced in the previous chapters in order to generate (parts of) the language-specific symbol table infrastructure based on (marginal) extensions of the MontiCore grammar. It further describes non-invasive ways for customizing the generated infrastructure.

Chapter 8 describes the extensions of the technical infrastructure (introduced in Chapter 4) required for conducting language composition. Moreover, it describes how language composition impacts the symbol table creation (cf. Chapter 5) and the resolution process (cf. Chapter 6). Finally, we present a reference implementation for the symbol table structure of Java-like languages.

Chapter 9 summarizes the main results of the thesis and outlines research tasks for future work.

Chapter 2

Model-Driven Engineering and the MontiCore Language Workbench

This chapter introduces the model-driven engineering approach (cf. Section 2.1) and gives an overview of concepts and features of the MontiCore language workbench (cf. Section 2.2). The infrastructure of the present work extends MontiCore and further builds on lessons learned from the infrastructure developed by Steven Völkel [Völ11]. Essential parts of Völkel's work are elucidated throughout the respective chapters and there compared with the work of this thesis.

2.1 Model-Driven Engineering

Software is increasingly pervasive in many domains such as energy management, health care, intelligent transportation, smart homes, and logistics. Also, the complexity of software-intensive systems as well as their (physical and logical) heterogeneity has risen. The *problem-implementation gap*, i.e., the "wide conceptual gap between the problem and the implementation domains" [FR07] impedes the development of such complex software systems. In particular, this gap arises from the fact that the concepts used to implement complex software systems (e.g., via general purpose languages, GPLs) are lower than the concepts of the problem domain leading to the *accidental complexity*. Manually overcoming this gap with (traditional) implementation approaches is both time-consuming and labor-intensive [CCF+15].

Model-driven engineering (MDE)[1] aims at narrowing down the problem-implementation gap and thereby the accidental complexity by utilizing formal models as first-level development artifacts for, among others, designing, automated analyzing as well as synthesizing, deploying and maintaining software systems (cf. [Sel03, Sel06]).

The term "model" is not commonly defined in computer science [Sei03, Kü05]. There is, however, widespread recognition—as in the current thesis—that a *model* is an abstraction of the original it denotes by, for example, abstracting from unnecessary details [Rum16] (cf. [Sta73, HBvB+94, Bal00, BG01, Sei03, Küh06, Sch12]). Furthermore, models define specific aspects of a software system supporting separation of concerns.

[1] or Model-driven development (MDD)

Models can be expressed in various types of software languages, such as *domain-specific languages* (DSLs) and GPLs. In contrast to GPLs (such as Java and C#), a DSL is specialized for a designated domain and does not have to be executable. Some popular DSLs are HTML [HTM] for specifying web documents and SQL [SQL11] for database management. Throughout this thesis, we use the term *modeling language* (or solely *language*) to refer to any language (DSL or GPL) that is utilized in the context of modeling. For example, although Java is a GPL it can serve as an action language (cf. [OMG13]). In this context we consider Java classes as *Java models*. Multiple models of different modeling languages can be composed to define the whole software system. This topic is elucidated in Chapter 8.

In particular, a language consists of concrete syntax, abstract syntax, context conditions and semantics [HR00, HR04]. The *concrete syntax* determines the representation of a language and can be textual or graphical (or a mixture of these) [GKR+07]. Textual languages typically rely on tools such as ANTLR [PQ95, Par07], SableCC [GH98], and ASF+SDF [BHD+01] for producing parsers (cf. [Völ11]). In contrast, graphical languages are built with frameworks such as the Eclipse Graphical Modeling Project (GMP) (or Eclipse Graphical Modeling Framework, GMF) [www16m, Gro09], Microsoft DSL Tools [CJKW07], or MetaEdit+ [KLR96, Met16] to provide a graphical notation (e.g., boxes and lines).

The *abstract syntax* of a language represents its internal structure consisting of the *essence* of that language [CvdBCR15]. It omits semantically irrelevant information, such as syntactic sugar. The abstract syntax can to some extent be generated from a grammar definition as, for example, in the MontiCore language workbench (cf. Section 2.2). Some approaches separate the specification of abstract and concrete syntax to allow that "several concrete syntaxes can be mapped to one abstract syntax" [HHJ+08]. Alternatively, the abstract syntax can be specified via meta-models using, for example, the Eclipse Modeling Framework (EMF) [SBPM09]. In textual languages, the parsed models are represented by a tree structure called the *abstract syntax tree* (AST). In the current thesis, the abstract syntax consists of both the AST and the symbol table (cf. Section 3.8).

Context conditions [HR00] are rules (i.e., boolean predicates) operating on a languages abstract syntax to statically check its consistency (cf. [ALSU06]). A model is said to be *well-formed* if it fulfills all context conditions of its language. Context conditions can also employ type systems [Car97] for the well-formedness check. For this, attribute grammars [Knu68] can be applied, especially, to resolve the type of complex expressions.

The *semantics* determines the meaning of a language and can be specified via mathematical constructs (denotational semantics), abstract machines (operational semantics), or transformations to an existing language (translational semantics) [Kle07]. Hans Grönniger [Grö10] presents a framework for semantics of DSLs mapped to a theorem prover.

Code generators play an essential role in MDE as they embody the knowledge for bridging the problem-implementation gap by transforming "a set of input files to a set of executable output files" [Sch12]. In particular, two kinds of transformations can be

distinguished, namely model-to-model (M2M) and model-to-text (M2T) [CH03, CH06]. M2M transforms an input model to an output model typically based on the abstract syntax (e.g., meta-model), and therefore, requires the abstract syntax of both models. The input and output models can be specified by the same language or by different languages. In contrast, M2T transforms the input model to a string output (of a GPL), which does not have to adhere to a specific abstract syntax. Popular examples are template-based code generators build on engines such as FreeMarker [www16d] and Velocity [GC03].

MDE has already been successfully applied as in, e.g., [WWM+07, Rai05, KR05, SV06, VRKS13]. A famous model-driven approach is the Model Driven Architecture (MDA) [OMG03] of the Object Model Group (OMG). The main idea of MDA is to develop abstract platform-independent models and successively transform them to more concrete platform-dependent models. For this, MDA provides three kinds of models, namely a computation independent model (CIM), a platform independent model (PIM), and the platform-dependent model (PSM). By applying these, MDA, among others, aims at increasing the portability and quality of the software system. MDA builds upon several other standards and specifications of OMG, such as the Unified Modeling Language (UML) [OMG15c], the Meta Object Facility (MOF) [OMG15b], the XML Metadata Interchange format (XMI) [OMG15d], and the model transformation language family QVT [OMG15a].

2.2 The MontiCore Language Workbench

In order to support the development of modeling languages many authors suggest so called language workbenches (e.g., [Fow10, Gho10, VBD+13]). A *language workbench*, among others, aids in specifying a language's meta-model (or grammar) and also provides mechanisms for analyzing, manipulating and transforming a language's models.

This thesis builds on MontiCore, a language workbench for textual DSLs which has been developed at the chair of Software Engineering at the RWTH Aachen University. Recently, MontiCore has the version 4. The remainder of this section introduces relevant aspects of this version (referred to as MontiCore 4 or solely MontiCore). Detailed information about previous versions and general concepts—which also partly apply to MontiCore 4—can be found in [GKRS06, GKR+06, KRV07a, KRV07b, KRV08a, KRV08b] as well as in the PhD theses of Holger Krahn [Kra10], Steven Völkel [Völ11], and Martin Schindler [Sch12].

MontiCore 4 is a lightweight, API-based language workbench developed with Java. It employs Apache Maven [www16a] for build management. In MontiCore the concrete syntax of a modeling language is defined via a grammar. The grammar also specifies the AST which together with the symbol table embodies the abstract syntax of a modeling language (cf. Section 3.8). Same as previous MontiCore versions, type systems are based on visitors and Java. Semantics is realized by means of translations (i.e., translational semantics [Kle07]) to target code via code generators.

In contrast to previous versions, MontiCore can be used (to some extent) in a functional manner, i.e., all needed arguments are explicitly given when objects are instantiated or methods are called. Particularly, this means that main components such as parsers and the generator engine do not rely on implicit infrastructure classes in order to conduct their specific tasks. As a consequence, MontiCore's components are modular and easy to use. While in previous MontiCore versions a DSL tool (i.e., a tool for processing models of a DSL) is manifested in a technical class (named `DSLTool`), in MontiCore 4 a DSL tool exists only conceptually by defining a workflow of processing steps via the provided API.

Figure 2.1 depicts the general architecture of a DSL tool developed with MontiCore. The *workflow execution* component provides functionality to set and configure the workflows that have to be executed on the input models, such as parsing and context checks. The concrete execution order is specified with Groovy [KLK+15, www16f], a dynamic language developed for the Java platform. The *parser* transforms the textual input models to an internal AST structure. The AST (together with the symbol table, cf. Section 3.8) is the central structure of the model processing. The *function library* consists of common functionalities provided by MontiCore's runtime project[2], e.g., for logging and file handling. Finally, the *generator engine* (also part of MontiCore's runtime) produces artifacts for, e.g., code and reports. It is based on the open source, template-based FreeMarker engine [www16d].

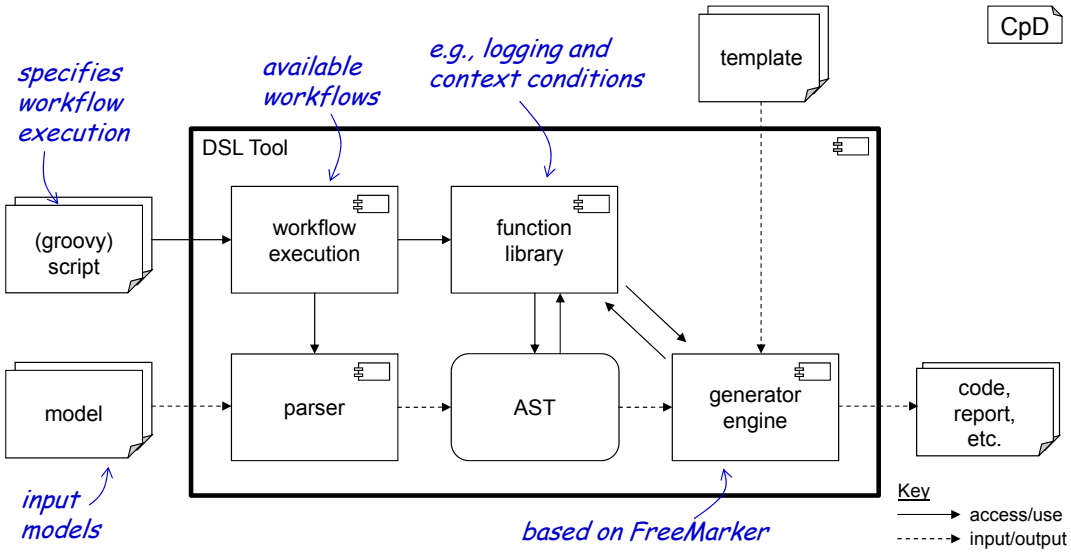


Figure 2.1: General architecture of a DSL tool in MontiCore.

[2] see `https://github.com/MontiCore/monticore/tree/master/monticore-runtime`

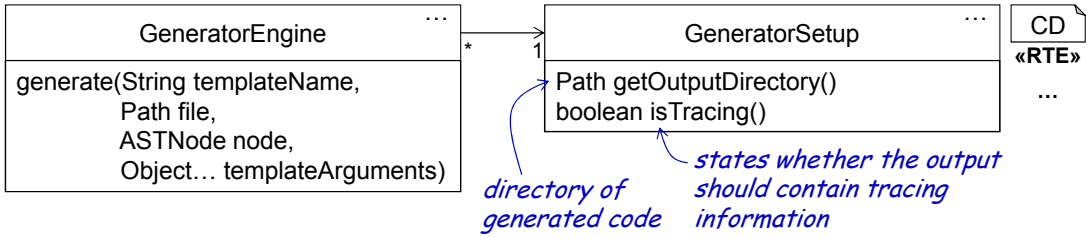


Figure 2.2: The `GeneratorEngine` class starts the generation process. It can be configured with the `GeneratorSetup` class.

MontiCore's generator project[3] itself is a DSL tool for processing grammars (as input models). The typical procedure for processing a MontiCore grammar is as follows:

1. First, some initialization is conducted, e.g., the infrastructure for loading grammars is initialized.
2. Grammars are loaded and the resulting AST is checked for well-formedness.
3. A parser is generated for each grammar using ANTLR.
4. The grammar AST is transformed to a class diagram AST (CD AST) which then is decorated with further methods.
5. Then, the CD AST serves as input for generating some parts of the language-specific infrastructures, i.e., AST classes, the visitor infrastructure, the symbol table infrastructure, and the context condition infrastructure.
6. Finally, some reports are generated.

This workflow is specified with the Groovy script listed in Appendix C.

2.2.1 Generator Engine

The `GeneratorEngine` class conducts the M2T transformation in MontiCore. The applied concepts are based on [Sch12]. As depicted in Figure 2.2, `GeneratorEngine` solely provides the method `generate` which starts the template-based generation with the template `templateName` using FreeMarker. The `node` parameter represents the model or model element which serves as input. The output is generated into the specified `file`. If no absolute path is given, the output will be stored into the `file` relative to the `outputDirectory` configured in `GeneratorSetup`. The last parameter of `GeneratorEngine`, i.e., `templateArguments` allows for an arbitrary number of

[3] see `https://github.com/MontiCore/monticore/tree/master/monticore-generator`

additional arguments that are passed to the template. The method `isTracing` of `GeneratorSetup` states whether the generated code should contain information that enables tracing back to the input AST.

2.2.2 MontiCore Grammar

MontiCore provides an extended grammar format based on EBNF which enables specifying productions for the lexer as well as for the parser. A production consists of a left-hand side (LHS) and also defines a right-hand side (RHS) except for interface, abstract, and external productions described below. MontiCore's grammar specifies both the concrete and (parts of) the abstract syntax [KRV07b], and therefore produces respective parsers and AST classes. Additionally, MontiCore allows for grammar as well as production inheritance.

Lexical Productions

A lexical production is specified by a regular expression. Listing 2.3 defines the `Name` production via the keyword `token`. Lexicals are mapped to strings in the respective AST node.

```
1 token Name =                                                    MCG
2   ( 'a'..'z' | 'A'..'Z' | '_' | '$' )
3   ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$' )*;
```

Listing 2.3: Lexical production `Name` of `Lexicals` grammar.

Terminals

Terminals are atomic elements specified in the RHS of a production between quotation marks, e.g., `"x"` in Listing 2.4 (line 2). Terminals are not part of the generated AST classes since the information is not semantically relevant.

There are, however, two exceptions. First, if the terminal is assigned a name, e.g., `type:"class"`, a respective string attribute is generated in the AST. Second, optional terminals, i.e., in alternatives or marked with `?`, between square brackets result in a boolean attribute in the AST class, e.g., `["initial"]?`.

Class Productions

Class productions are the default in MontiCore, and hence, do not require a dedicated keyword. They are called *class* productions since they are mapped to AST classes.

```
1 // use of terminal and lexical                                    MCG
2 A = "x" Name;
3 // optional nonterminal
4 B = A?;
5 // alternative nonterminals
6 C = D | E;
7 // arbitrary number of nonterminals
8 D = A*;
9 // at least one nonterminal
10 E = A+;
11 // explicit nonterminal name
12 F = z:A?;
```

Listing 2.4: Examples of atomic nonterminal operators, i.e., `?` (optional), `|` (alternative), `*` (any number), and + (at least one).

The LHS of a class production defines the nonterminal's name `NT`, leading to an AST class `AST`*NT*. The RHS defines the production's body consisting of any combination of lexicals, terminals and nonterminals. Listing 2.4 gives an overview of the atomic operators for the RHS. These can be combined to form more complex constructions.

Optional nonterminals (line 4) are specified via `?` and result in an optional field of the (generated) AST node. In this example, the AST node of `B`, namely `ASTB`, defines the field `a` of type `Optional<ASTA>`[4]. The name derived for the field is the nonterminal's name in lowercase. Similarly, alternatives specified via `|` result in optional fields (line 6). Hence, `ASTC` contains the fields `d` and `e` of type `Optional<ASTD>` and `Optional<ASTE>`, respectively. The `*` operator allows to specify an arbitrary number of nonterminals (line 8) and leads to a list in the respective AST node. Hence, `ASTD` has the field `List<ASTA> a`. This also applies for the + operator (line 10) which declares that a nonterminal must occur at least once. To change a fields name in the AST, MontiCore's grammar allows to explicitly specify a nonterminal's name. For instance, `F = z:A?` (line 12) leads to the field `Optional<ASTA> z` in `ASTF`. Furthermore, MontiCore 4 introduces the syntax `x:(A || ",")+` which is a shorthand for `x:A ("," x:A)*`.

Interface, Abstract, and External Productions

MontiCore's extended grammar format allows for two concepts known from object-oriented languages, i.e., the definition of interface and abstract productions (or nonterminals). Figure 2.5 shows an example of an *interface nonterminal* (left part) and the generated AST structure (right part). Production `I` defines an interface nonterminal via the keyword

[4] see java.util.Optional (since Java 8), `https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html`

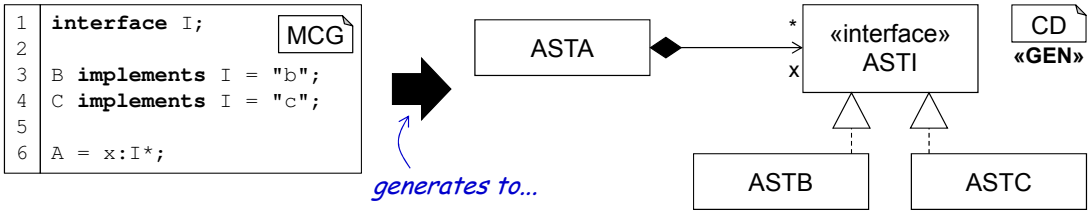


Figure 2.5: Interface nonterminal in MontiCore grammar.

`interface` (line 1). Same as interfaces in object-oriented languages, the body (i.e., the RHS) is omitted, hence, no concrete syntax is defined. The nonterminals `B` and `C` implement `I` and define their own RHS. Production `A` uses nonterminal `I` in its RHS (line 6), and that way includes all nonterminals implementing `I`, i.e., `B` and `C`.

As it can be seen on the right side of Figure 2.5, the nonterminal hierarchy is reflected in the class hierarchy of the respective AST nodes. That is, `ASTI` is an interface and subtyped by `ASTB` as well as `ASTC`.

Similarly, an *abstract nonterminal* is introduced via the keyword `abstract`, has no RHS, but results in an *abstract* AST node. Other nonterminals can extend it via `extends`, e.g., `B extends A` where `A` is the abstract nonterminal.

Finally, *external nonterminals* (marked with `external`) define extension points that—in contrast to interface and abstract productions—can only be specified in subgrammars. For each external nonterminal `NT` MontiCore produces an AST node interface `AST`*`NT`*`Ext`.

An external production is similar to a *slot* as defined in [HHJ+08], i.e., it can be replaced only once. In contrast, a *hook* can be replaced several times [HHJ+08], as non-external productions in MontiCore. Please note that Henriksson et al. use the term "replace", resulting from the invasive composition approach which conducts composition by *transformation* [Aßm03].

Component Grammar

A *component grammar* represents an incomplete grammar with designated extension points (i.e., external, interface, or abstract nonterminals), analogously to abstract classes in object-oriented languages. It is introduced via the keyword `component` before the `grammar` keyword.

Component grammars are useful for creating a library of tokens and nonterminals. The `Lexicals` grammar in MontiCore, for example, defines the token `Name`, which can be used by any grammar via inheritance. For component grammars, MontiCore produces AST classes but no parsers. Parsers are only generated for subgrammars if they are no component grammars themselves.

As already mentioned, external nonterminals may only be concretized in subgrammars. Hence, defining an external nonterminal always results in a component grammar. Same

is true for interface and abstract nonterminals that are not extended within the grammar they are defined in.

Grammar Composition

MontiCore employs grammar inheritance, among others, for (i) extending a grammar, e.g., with new productions, and for (ii) conducting grammar embedding, e.g., by binding an external production. In the current thesis these concepts are of particular interest since they enable language inheritance and language embedding (cf. Chapter 8), respectively.

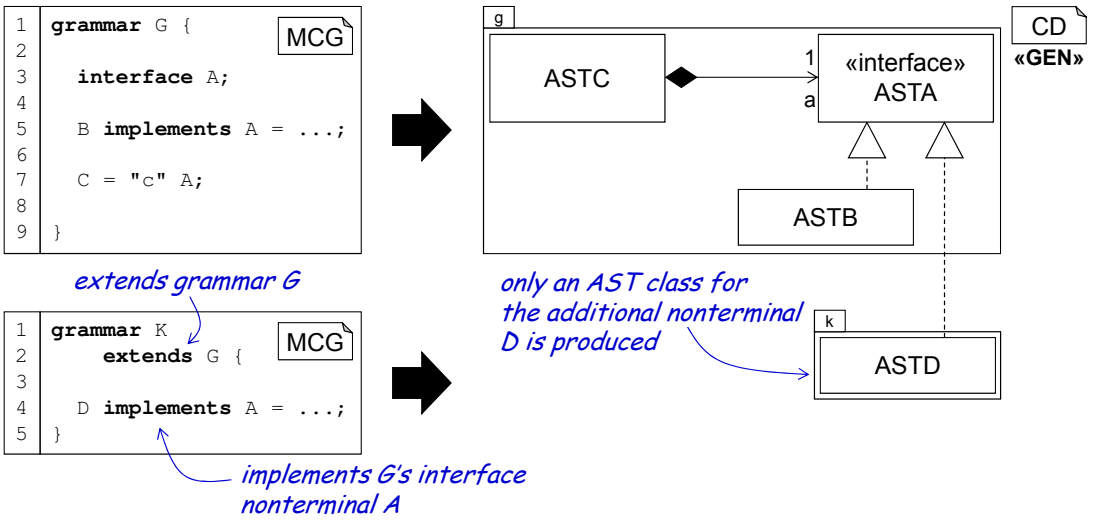


Figure 2.6: Example of grammar inheritance with respective AST classes.

Grammar inheritance allows for reuse and overriding[5] of productions of other grammars. The extending grammar inherits all productions of its supergrammars. Figure 2.6 shows an example of grammar inheritance. Grammar `G` defines the interface nonterminal `A` as well as the class nonterminals `B` and `C`. Grammar `K` extends `G` to reuse its nonterminals. `K` additionally introduces the nonterminal `D` which implements `A`. As it can be seen in the right part of Figure 2.6, only the AST class `ASTD` for the additional nonterminal in `K` is produced. The AST classes of `G` are completely reused.

In *grammar embedding* nonterminals of a grammar are embedded into designated extension points (e.g., external nonterminals) of another grammar. The former is the *embedded grammar* whereas the latter is the *host grammar*. Technically, MontiCore conducts grammar embedding via multiple grammar inheritance where a *glue grammar* extends both the host and the embedded grammar. In the example shown in Figure 2.7,

[5] Production overriding can be employed in order to actually restrict the "extended" grammar, resulting in *grammar restriction* (cf. [EGR12]).

G and E are two independent grammars (e.g., specified by different language engineers). Since G defines the external nonterminal A (line 3), it is a component grammar. The right part of Figure 2.7 illustrates the emerging AST structure.

The third grammar K is the glue grammar which serves as configuration for embedding the nonterminal D of grammar E into nonterminal A of grammar G. For this, K inherits both G and E (line 2). Next, it specifies a production where A is the LHS and D the RHS (line 4). As it can be seen in the right part of Figure 2.7, the AST class ASTA is produced for the K grammar and extends the ASTAExt interface generated from the external nonterminal A of grammar G. Additionally, ASTA contains ASTD of grammar E.

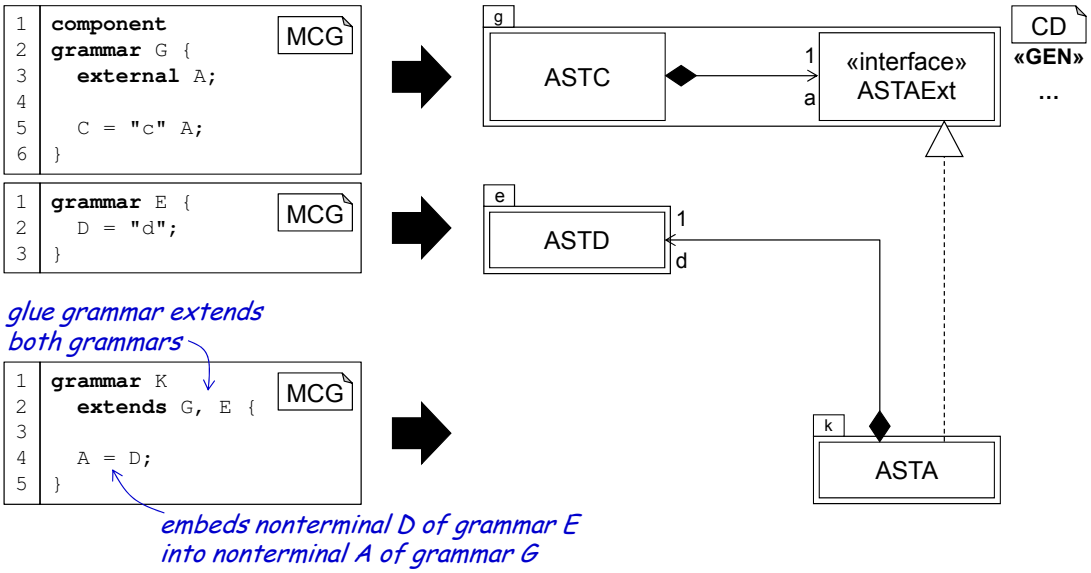


Figure 2.7: Example of grammar embedding with respective AST classes.

2.2.3 Generated Parsers

MontiCore uses ANTLR [PQ95, Par13] to generate a parser from a grammar. The parser provides methods for each nonterminal (i.e., the respective AST node) to enable parsing of a whole model as well as model parts.

As exemplified in Figure 2.8, the parser GParser is generated from the grammar G. It provides three parsing methods for each nonterminal enabling different possibilities for the input, i.e., path to the file name, a Reader[6], or the content directly as a string. The return value is wrapped in an Optional object which is present in case the parsing was successful.

[6] see java.io.Reader, https://docs.oracle.com/javase/8/docs/api/java/io/Reader.html

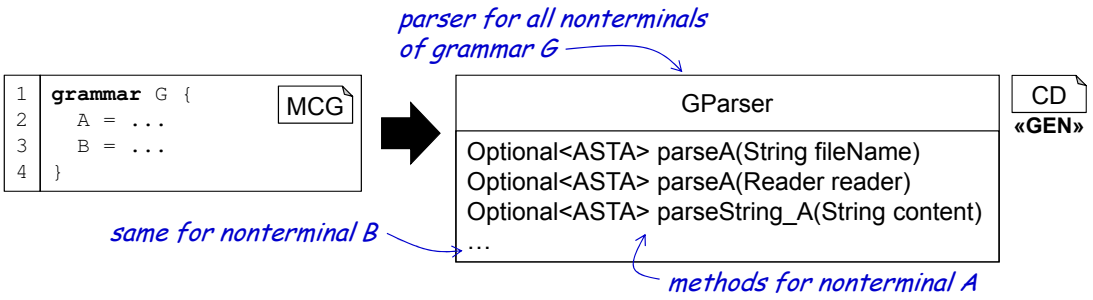


Figure 2.8: Example of a Generated Parser from a Grammar.

2.2.4 Generated Visitor Infrastructure

Model processing in MontiCore can be conducted by traversing the AST using the Visitor Pattern [GHJV95]. While previous MontiCore versions follow a generic reflection-based approach, MontiCore 4 employs a generative approach for producing statically type-safe visitor interfaces. This section gives a brief overview of MontiCore's visitor concept. A detailed introduction and discussion can be found in [HMSNRW16].

MontiCore produces three kinds of visitors for each grammar, namely, a default visitor, an inheritance visitor, and a delegation visitor. Figure 2.9 shows an example of a default and an inheritance visitor.

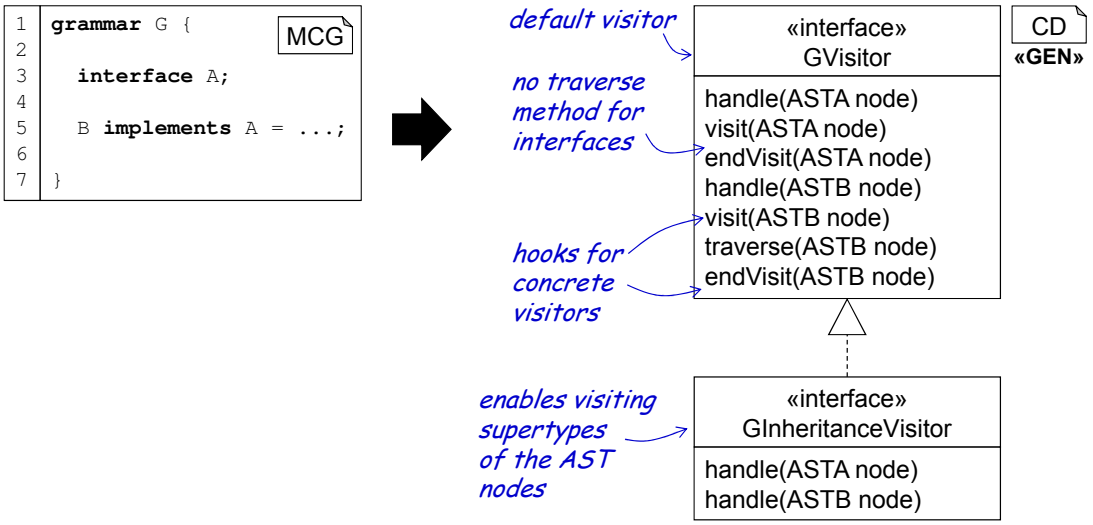


Figure 2.9: Example of a default visitor and an inheritance visitor generated from the grammar.

```java
default void handle(ASTB node) {
  getRealThis().visit(node);
  getRealThis().traverse(node);
  getRealThis().endVisit(node);
}
```

Java «GEN»

Listing 2.10: `handle` method of a default visitor. First, the node is visited. Then, its child nodes are traversed. Finally, the `endVisit` method is invoked on the node. The `getRealThis` method enables reuse of the visitor.

As it can be seen in Figure 2.9, the default visitor `GVisitor` provides four methods for each (non-abstract) AST node. The `handle` method determines the order of the other three methods, as shown in Listing 2.10. The `visit` and `endVisit` methods are processed before and after a node is traversed, respectively. The `traverse` method specifies the traversal strategy of the child nodes which by default is conducted using a depth-first approach. The `getRealThis` method invoked in the `handle` method (cf. Listing 2.10) is important for visitor reuse and is described below together with the delegator visitor. Since non-class productions do not specify a RHS, no `traverse` method is generated for them.

The default visitor calls the most specific method for each AST node. In order to omit type introspection, MontiCore simulates double dispatching (cf. [HMSNRW16]). For this, an interface `ASTGNode` is generated for each grammar `G` which solely provides the `accept(GVisitor)` method to start the traversal on any visitor of the grammar. Each AST node of `G` subtypes `ASTGNode`.

In order to process supertypes of the AST nodes, MontiCore also produces an inheritance visitor, e.g., `GInheritanceVisitor` (cf. Figure 2.9). `GInheritanceVisitor` subtypes `GVisitor` and overrides each `handle` method as shown in Listing 2.11.

```java
default void handle(ASTB node) {
  // calls visit method of ASTA, i.e., ASTB's super type.
  getRealThis().visit((ASTA)node);
  getRealThis().visit(node);
  getRealThis().traverse(node);
  getRealThis().endVisit(node);
  // calls endVisit method of ASTA
  getRealThis().endVisit((ASTA)node);
}
```

Java «GEN»

Listing 2.11: `handle` method of an inheritance visitor. Additionally to the default visitor, it invokes the `visit` and `endVisit` methods of each supertype of the AST node.

Besides the methods already called in the respective `handle` method of `GVisitor` (cf. Listing 2.10), the overridden `handle` method in `GInheritanceVisitor` additionally calls the `visit` (line 3, Listing 2.11) and `endVisit` (line 8) methods of `ASTA`, i.e., the supertype of `ASTB`.

Visitors for Grammar Inheritance

Visitors of supergrammars can be reused via subclassing. Figure 2.12 shows an example. The top part depicts the generated visitor infrastructure for the grammars `G` and `K` shown in Figure 2.6 (on page 17). As it can be seen in Figure 2.12, `KVisitor` extends `GVisitor` and solely adds additional methods for `ASTD`.

Similarly, a concrete visitor of `K` extends a concrete visitor of `G` in order to reuse the defaults for AST nodes of grammar `G` and implement the additional methods of `KVisitor` (bottom part of Figure 2.12).

Please note that `GVisitor` does not provide a specific `visit` method for `ASTD` (since nonterminal `D` is introduced in its subgrammar `K`), and thus, can only handle `ASTD` nodes in the more general method `visit(ASTA)`. This, however, can lead to undesired results, and therefore, AST nodes should only be processed by visitors of their own language (cf. [HMSNRW16]).

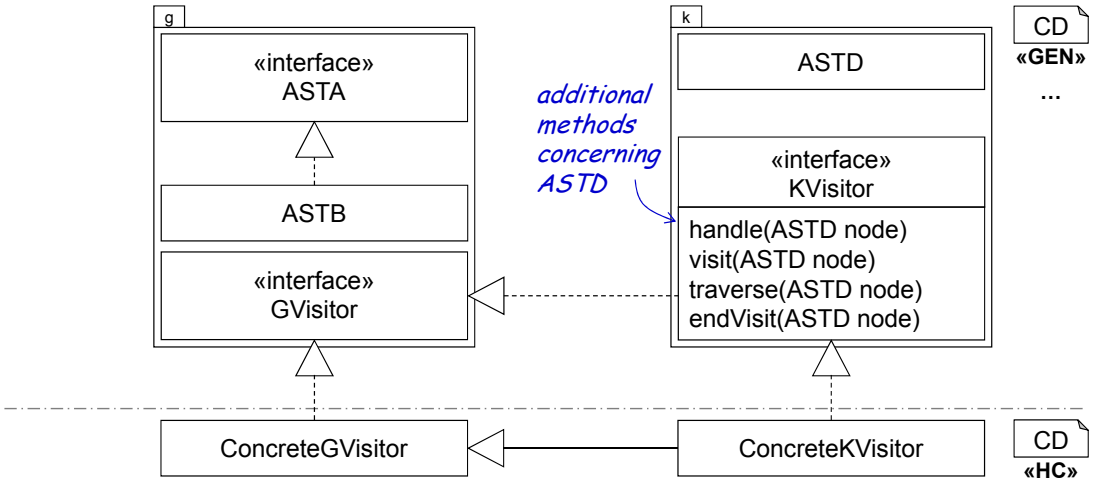


Figure 2.12: Example of visitor inheritance.

Visitors for Grammar Embedding

MontiCore conducts grammar embedding via multiple inheritance (cf. Section 2.2.2). Since Java only allows for single inheritance, the respective visitors are reused via delegation. For this, MontiCore produces a *delegator visitor* for each grammar.

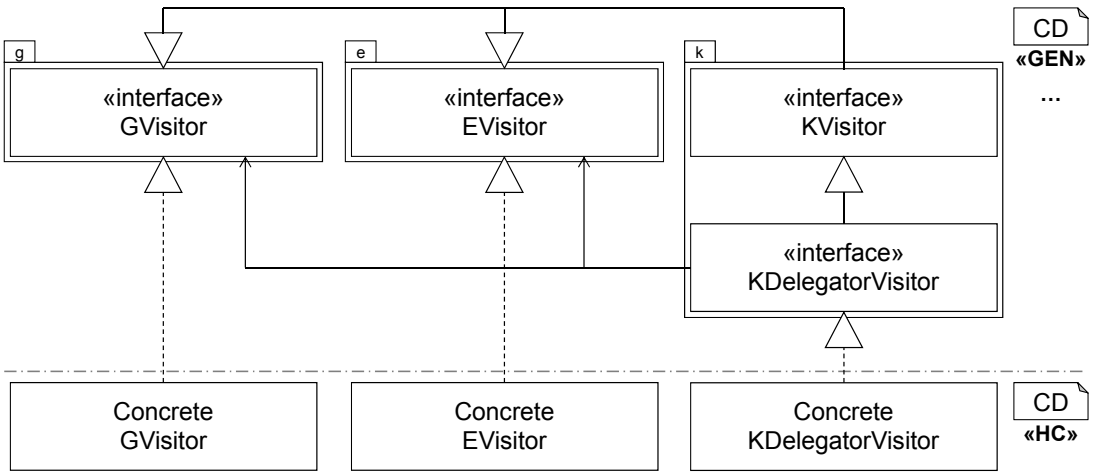


Figure 2.13: Example of a concrete delegator visitor.

Figure 2.13 shows the emerged delegator visitor infrastructure for the three grammars introduced in Figure 2.7 (on page 18). The delegator visitor `KDelegatorVisitor` implements `KVisitor` which itself extends the visitors of `K`'s supergrammars, i.e., `GVisitor` and `EVisitor`. Moreover, `KDelegatorVisitor` delegates to an instance of either visitor, depending on the AST node to be processed.

For this, it is essential that the visitors make use of MontiCore's *realThis* pattern (cf. [Sch12]). In short, this means the keyword `this` should be omitted in the visitors, instead the generated method `getRealThis` is to be used, as already shown in Listing 2.10 and Listing 2.11. This allows for managing the handling of a node within the delegator visitor and also share states between the single (independent) visitors when composing them.

By default, `getRealThis` points to the current visitor (i.e., `this`) and is changed in the `setRealThis` method of a delegator visitor. The method is implemented in such a way that it transitively sets the correct `realThis` reference of the visitors it delegates to which can be composed themselves.

Listing 2.14 shows the initialization of a concrete delegator visitor of grammar `K`. The visitor extends `CommonKDelegatorVisitor` (not shown in Figure 2.13) which provides a default implementation of the `KDelegatorVisitor` interface. The constructor sets the delegates, that are, the concrete visitors of `G` and `E` as well as the concrete visitor of `K` in order to handle AST nodes of `K`. For all these visitors the `realThis` reference is set to be the `ConcreteKDelegatorVisitor` instance.

2.2.5 Generated Context Condition Infrastructure

For each nonterminal `NT` of a grammar `G` MontiCore produces an interface named `GAST`*`NT`*`CoCo`, as shown in Figure 2.15. The interface provides one `check` method with

```
1 public class ConcreteKDelegatorVisitor                    Java
2     extends CommonKDelegatorVisitor {                     «HC»
3
4   public ConcreteKVisitor() {
5     setGVisitor(new ConcreteGVisitor());
6     setEVisitor(new ConcreteEVisitor());
7     setKVisitor(new ConcreteKVisitor());
8   }
9 }
```

Listing 2.14: Configuration of a concrete delegator visitor.

the AST node as its parameter. Concrete context conditions must implement the interface to define well-formedness checks for the represented model element.

Additionally, the `GCoCoChecker` class is generated (cf. Figure 2.16) which allows to group context conditions and process them on respective AST nodes. In order to register context conditions, it provides a dedicated `addCoCo(ASTNT)` method for each nonterminal `NT`. `GCoCoChecker` is a visitor (it subtypes `GVisitor`) and that way enables to conduct the registered checks on a given `ASTNT` node via the method `visit(ASTNT)`. Moreover, being a visitor, `GCoCoChecker` also allows for checking AST subtrees. For this, it provides the method `checkAll(ASTGNode)` which starts the check with any AST node of the `G` grammar and continues with its subnodes (using the visitor's `traverse` method).

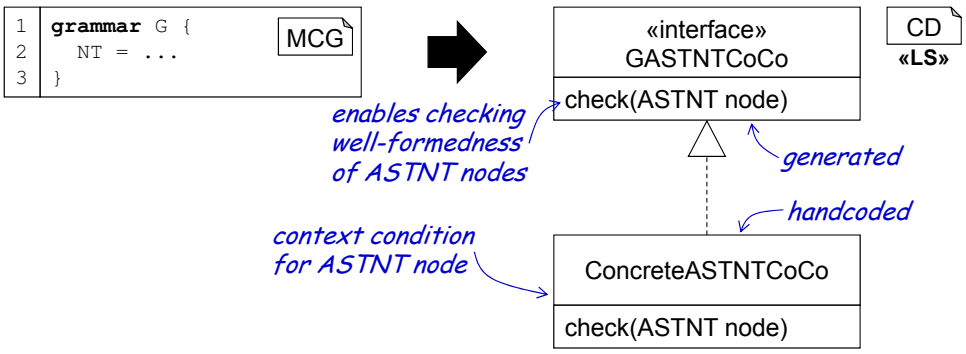


Figure 2.15: Example of a generated context condition interface for a specific AST node.

2.2.6 Paths in MontiCore

Based on Apache Maven's standard directory layout [www16a], MontiCore provides several paths for the input artifacts and an output path for the generated artifacts. The

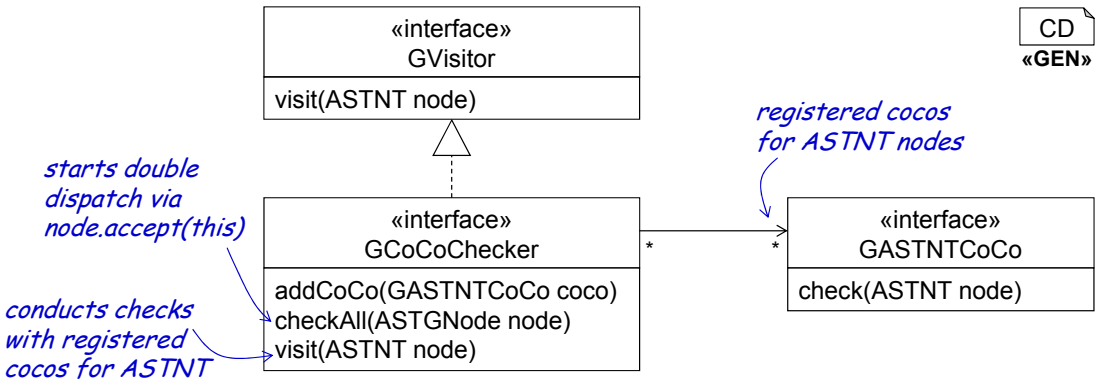


Figure 2.16: Generated context condition checker for the grammar shown in Figure 2.15.

following list summarizes these paths. The information in parentheses refers to the default paths when running MontiCore standalone (e.g., via command line).

Grammar Paths: Paths to input grammars. The default is `src/main/grammars` (standalone: `grammars`).

Handcoded Java Paths: Paths to handwritten Java code. This, above all, enables efficient integration of handwritten and generated code, as described in Section 7.14. By default, the path is `src/main/java` (standalone: `java`), i.e., the same path as for the productive code.

Model Paths: Model dependencies are resolved in these paths. By default, it contains the grammar paths described above.

Template Paths: Paths to templates (i.e., `.ftl` files) that are to be integrated into the code generation process. Standardly, it is `src/main/resources` (standalone: `resources`).

Output Path: The path in which the generated code is produced. By default, it is `target/generated-sources/monticore/sourcecode` (standalone: `target/sourcecode`). For a grammar `G` in a package `p`, MontiCore additionally generates a language's components in dedicated subdirectories of `p/g/`, that are:

_ast: contains the generated AST classes.

_cocos: contains the generated context condition infrastructure.

_parser: contains the generated parser infrastructure.

_symboltable: contains the generated symbol table infrastructure (cf. Chapter 7).

_visitor: contains the generated visitor infrastructure.

All subdirectories statically depend on `_ast`. Moreover, `_symboltable` depends on `_parser` and `_visitor`.

MontiCore provides the class `MontiCoreConfiguration` which enables access to the specified paths.

2.2.7 Licensing

MontiCore 4 (including the results of this thesis) is open source and hosted in its own repository (see [www16j]) in GitHub [www16e]. While the Java code of MontiCore's runtime and generator projects is available under the *GNU LGPL V3* license [GNU07], templates and generated code are available under the *BSD-3-Clause* license [BSD99].

In summary, that means:

- Developers are free to modify and extend the generated code as needed.
- It is not necessary to mention that MontiCore was used, e.g., for generating code.
- It is allowed to use the generated code in closed source and commercial software.
- Only changes that *directly concern MontiCore* must be published under LGPL.

2.2.8 Related Language Workbenches

The following comparison of language workbenches is partly based on the Language Workbench Challenge[7] [EvdSV+13, EvdSV+15] where a more comprehensive discussion can be found. Moreover, for a comparison of previous MontiCore versions with other language workbenches, please refer to [GBU08, PP08, Kra10, Völ11, Sch12].

Same as MontiCore, the language workbenches Xtext, Spoofax, SugarJ, and Rascal use a *textual* notation, but, in contrast to MontiCore, all are built on the Eclipse IDE [www16c]. Xtext [EB10] is EMF-based [SBPM09, www16b] and employs Ecore as its underlying meta-model[8]. Same as MontiCore, Xtext is grammar-based and utilizes the ANTLR parser generator. Also, Xtext's grammar enables the definition of both the concrete as well as the abstract syntax. Xtext provides the Java-like language Xtend for specifying the code generation.

Spoofax [KV10, www16n] is a language workbench for textual languages based on Eclipse. It exploits several declarative meta-languages such as SDF [Vis97] for defining context-free grammars and Stratego [BKVV08] for code generation as well as AST transformation via rewrite rules. SugarJ [ERKO11, www16l] combines some of these meta-languages together with a type system DSL [LE13] into a base language, which

[7] see Language Workbench Challenge website `http://www.languageworkbenches.net/`

[8] Since recently, Xtext is also available for the IntelliJ IDE (see `http://www.xtext.org`).

can be extended via a library-based approach. Sugarclipse [EKR+11] is an IDE for SugarJ based on Spoofax. Same as Spoofax and SugarJ, Kermeta employs dedicated meta-languages to specify the different concerns of a language [JBF11, JCB+15]. The abstract syntax (or meta-model) of a language can be defined with a meta-language that is based on EMOF [OMG15b]. Another meta-language based on the Object Constraint Language (OCL) [OMG14] allows the language engineer to define a language's static semantics (i.e., a set of well-formedness rules). The dynamic semantics (or behavioral semantics) is defined via the Kermeta language. In contrast to Spoofax, SugarJ, and Kermeta, MontiCore specifies semantics (static and dynamic) in Java.

In *projectional* language workbenches such as MPS [VS10, www16h], Intentional Domain Workbench [SCC06], Whole [Ric05], and Más [EvdSV+13, Mas16] (following the trend of cloud-based tools [GR16]) the AST is the central artifact of editing. This essentially means that neither a grammar nor a parser exists. In contrast, the AST is manipulated directly and projected onto one or more notations. That way, both textual as well as graphical (and also tabular) notations can be used. Some language workbenches do not rely on a single approach. Onion, for example, combines textual parsing with projectional editing [EvdSV+13].

Graphical language workbenches, such as MetaEdit+ [KLR96, Met16], Microsoft DSLTools [CJKW07], and GME [LMB+01] focus on the development of graphical modeling languages. AToMPM [SVM+13] is a web-based open-source framework for the development of graphical languages. A language's abstract syntax is specified with a dedicated meta-model whereas models of other meta-models can be linked as well. AToMPM's view-based approach allows different users to work collaboratively on the same model. AToMPM enables each user to utilize her own concrete syntaxes for the same DSL.

Similar to MontiCore, the workbenches Xtext, Onion and Más rely on GPLs for specifying other aspects than the abstract syntax (e.g., code generation); Xtext uses both Java and Xtend, and Onion uses C#. Rascal [KvdSV09, www16k] relies on a single language which combines several domain-specific features such as grammar definition, AST traversal, and code generation. Most language workbenches enable code generation via M2T or M2M (or both). In contrast, the workbench Ensō [vdSCL14] focuses on model interpretation.

Same as MontiCore, JastAdd [HM03, EH07] employs a context-free grammar (called abstract grammar) to declaratively specify the syntax of the language. From this, JastAdd generates Java classes representing the AST and providing methods for manipulating and accessing (e.g., traversing) it. Nonterminals result in abstract classes (same as external, interface, and abstract nonterminals in MontiCore) while productions lead to concrete (sub-)classes. The AST classes can further be augmented via so-called inter-type declarations which allow for adding additional methods in an aspect-oriented manner [KHH+01]. That way, JastAdd also adds synthesized and inherited attributes (as well as equations) [Knu68] to the AST in the form of Java methods. The implementation

of those methods again is Java-based. In order to prevent unexpected behavior, those methods must not have any side-effects like mutating the AST node.

The Neverlang framework [VOSC14, VC15] follows a feature-oriented approach for modular language development. Features are elements of a language (e.g., a `while` construct), and can be processed in several evaluation phases, such as type checking and code generation. A language component provides an implementation for a specific feature, i.e., for its syntax and related semantics. Since a language component is self-contained, it can be easily reused in various languages (with similar feature requirements).

Chapter 3

Concepts and Elements for Symbol Management

In this thesis, we focus on *block structured*, *lexically scoped* (or statically scoped) languages since "[...] all important programming languages today permit the use of blocks [...]" [GM10] and "[m]ost languages [...] use static scope" [ALSU06].

Block structured [CL83, ALSU06, Seb08, GM10] means, that the program (within a file) is structured hierarchically by so-called blocks, which group declarations and statements [ALSU06]. Algol-60 [BBG+63] was the first block structured language, followed by more modern languages, such as C [KR88], Java [GJS+14], and C# [ECM06].

In lexically scoped languages the visibility of the defined elements can be determined "by looking at the program" [ALSU06], hence, before the runtime [Sco09]. In contrast, the visibility of elements in dynamically scoped languages (such as Logo, bash, PowerShell, and Emacs lisp) can only be determined when the program executes, i.e., during runtime [Wat04]. A major advantage of lexically scoped languages is that the well-formedness can be checked statically, for example, by compilers.

A key concept of software languages is the definition of new elements with names and their explicit use through these names (cf. [Wat04]). The underlying infrastructure often is a symbol table, which enables to retrieve information about a specific model element, called name resolution. According to Neron et al. "the basic concepts of resolution reappear in similar form across a broad range of lexically-scoped languages" [NTVW15]. However, both the literature as well as the software languages often use different terms for the same concepts and also same terms for different concepts (sometimes even inconsistently) [Str00, Völ11]. Hence, the contributions of this chapter are as follows:

1. It introduces *features and concepts* many block structured, lexically scoped software languages have in common, in particular, resulting from experiences gained from many tools and languages developed with Völkel's infrastructure [Völ11] (on top of previous MontiCore versions [Kra10]), such as the UML/P languages [Sch12, Rum16], JavaScript and TypeScript languages for MontiCore [Nes13], the RoboTask language family [HMSNR+15a], software categories languages [NN14, MSNR15a, MSNR15b], as well as MontiArc [HRR12, www16i] and its extensions MontiArc^{HV} [HRR+11], Δ-MontiArc [HRRS11, HKR+11], MontiArcAutomaton

[RRW12, RRW13a, RRW13b, RRW14], cloudADL [NPR13], and MontiSecArc [HHRW15]. The features and concepts also serve as *requirements* for subsequent chapters. Although the current thesis introduces common language features by the example of the general-purpose language (GPL) Java, it mainly focuses on DSL engineering. A single DSL usually does not need all the features presented in this thesis. In contrast, Java as a GPL is well known[1] and has many relevant features, such as shadowing, inheritance, packaging, top-level elements (e.g., classes), and inner elements (e.g., fields).

2. It gives a *clear definition* for each of those concepts[2] as well as
3. a *simple graphical notation* which then is used throughout this thesis.

Chapter Outline

Since the concepts are partly intertwined, the following Section 3.1 first gives a brief introduction to the core concepts of a symbol table as understood in the current thesis by the example of the Java language. Section 3.2 presents different forms of names in software languages. The subsequent Sections 3.3 – 3.8 define the core concepts and elements of the symbol table used in the current thesis, such as symbols and scopes. Next, Section 3.9 presents a simple graphical notation for some of the previously introduced elements. Section 3.10 defines encapsulated, imported, exported, and forwarded symbols which determine the reachability of specific model elements. Finally, Section 3.11 gives a method for finding candidates for symbols and scopes.

3.1 Introductory Example

The excerpt from the `java.lang.System`[3] class (cf. Listing 3.1) defines overall six elements, called *declared* entities in Java [GJS+14]:

- the package declaration `java.lang` (line 1)
- the class declaration `System` (line 3)
- the field declaration `props` (line 5)
- the method declaration `getProperty` (line 7)
- the parameter declaration `key` (line 7)
- the local variable declaration `sm` (line 9)

[1] According to the TIOBE index, Java is the most popular programming language of the years 2015 and 2016 (see `http://www.tiobe.com/tiobe_index`).
[2] Some of the definitions are already published in [HLMSN+15a, HMSNR15b].
[3] see `https://docs.oracle.com/javase/8/docs/api/java/lang/System.html`

```
1 package java.lang;                                          Java
2 // ...                                                    «MODEL»
3 public final class System {
4   // ...
5   private static Properties props;
6   // ...
7   public static String getProperty(String key) {
8     checkKey(key);
9     SecurityManager sm = getSecurityManager();
10    if (sm != null) {
11      sm.checkPropertyAccess(key);
12    }
13    return props.getProperty(key);
14  }
15  // ...
16 }
```

Listing 3.1: Excerpt from the `java.lang.System` class.

Each of these entities has a name to be identified and referred to from other parts of the program. For example, the variable `sm` is *defined* in line 9 and *used* in lines 10 and 11. In contrast, the return expression (line 13) does not have a name, and thus, cannot be referenced.

Depending on its *kind*, each entity embodies specific (meta-)information. The *class* `System`, for example, is `public` and `final`. Also, it contains the *field* `props` (line 5) and the *method* `getProperty` (line 7), which has the return type `String` and a parameter `key`. Furthermore, a program entity may only be defined in dedicated areas. A package, for instance, may only be declared outside the class body (analogously for enum or interface). In contrast, fields and methods must be declared within a class body. We call those areas where the different program entities may be defined *scopes*. Scopes affect the *visibility* of the containing entities. The parameter `key` (line 9), for example, is only visible within its enclosing method scope (lines 7–14). Moreover, scopes can be *nested*—the method scope is inside the class scope—leading to a hierarchical scope structure.

The current thesis uses terms from the MDE world. That means, instead of *program entity* [CWW80], we write *model element* (or just *element* if the meaning is clear). In textual software languages usually different kinds of model elements can be defined each containing its specific information. If those elements have a name, we call them *named model elements*. In Java, those elements are called *declared* entities [GJS+14]. Further synonyms (depending on the kind of language) are *named object* [GJR79], *denotable object* [GM10], *named entity* [GJR79, CL83], *entity* [WCW88], *declared object* [GJR79], and *object* [Sco09].

3.2 Names in Software Languages

Names[4] are a crucial part of textual software languages [GM10]. They enable defining (or declaring) model elements and referring to them. Associating (or binding) a name to the corresponding element definition is called *name resolution* (or name binding) (cf. [GM10, KKWV13, NTVW15]).

Furthermore, expressive names help improving the readability of a model (cf. [Wat04]). Therefore, some software languages state naming conventions (e.g., [GJS+14]). Also, names play an essential role when composing models (i.e., name-based model composition) [Rum13, HR13] (cf. Chapter 6 and Chapter 8).

A model element usually has (at least) two forms of names: a simple name and a qualified name. The *simple name* [HLMSN+15a] (or *unqualified name* [Che05, Völ11], *single identifier* [GJS+14]) is directly stated by the developer, such as `System` (line 3) and `props` (line 5) in Listing 3.1 and is unique within the defining scope. The *qualified name* [GJR79, ALSU06, Völ11, HLMSN+15a] (or *fully qualified name* [Che05, NTVW15]) uniquely identifies the entity throughout the system. Often, the qualified name is a composition of the names of the enclosing elements. For example, the qualified name of the class `System` in Listing 3.1 is `java.lang.System`, whereas `java.lang` is the qualified name of its enclosing package and `System` the class' simple name. Analogously, the qualified name of the field `props` is `java.lang.System.props`, with `java.lang.System` being the qualified name of its enclosing class and `props` its simple name[5]. We use the terms "fully qualified name", "qualified name", and "full name" interchangeably throughout this thesis.

Lastly, a combination of qualified and unqualified names can exist, namely *partially qualified names* (or partial names) [Che05, NTVW15]. For example, `System.props` is the partially qualified name of the field `props` in the `System` class (cf. Listing 3.1).

3.3 Symbols and Symbol Kinds

In order to access information of model elements (e.g., during generation time), their information is stored in the symbol table (cf. Def. 3.15) where the name is mapped to the associated information. The whole symbol table entry is called *symbol* (cf. [Par10, Völ11]) and is defined as follows [HLMSN+15a, HMSNR15b]:

[4] Following Strachey [Str00], we use the terms "name" and "identifier" interchangeably, in contrast to for example [ALSU06, GJS+14].

[5] Please note that a Java field does not have a qualified name [GJS+14]. Instead, referring to `java.lang.System.props` evaluates to the type of `props`, i.e., `Properties` (determined by the type system). Hence, information regarding the field (such as modifiers) cannot be retrieved. However, the language engineer needs those meta-information, for example, to develop well-formedness checks (cf. Section 3.8).

Definition 3.1 (Symbol (Definition))**.** *A* symbol definition *(or short* symbol*) contains all essential information about a named model element. It has a specific kind depending on the model element it denotes. A symbol is defined exactly once.*

While a symbol always has exactly one name, a named model element is not necessarily represented by a symbol. Symbols are stored in a symbol table (cf. Def. 3.15). A symbol (definition) as defined in Def. 3.1 comprises both the name of a model element and its associated information, and thus, we do not use the descriptive term "symbol table entry" as in, for example, [ALSU06] and [Völ11].

In the literature these two aspects, i.e., a name and its associated information, are usually separated, following the table approach of classical symbol tables which map a name to its associated information (e.g., [ALSU06, GM10, HR13]). Neron et al., for example, use the term *declaration* (or binding occurrence, cf. [Wat04]) that "introduces a name" [NTVW15] for, e.g., a variable. Similarly, Wolf et al. use the term *declaration* which "introduces an entity and associates an identifier (name) with that entity" [WCW88].

A symbol for a Java class, for example, contains information about, among others, the class' name, its members (i.e., fields and methods), superclass, and implemented interfaces. In contrast, a field symbol contains the type and name of the field element. Consequently, which information a symbol contains depends on its kind, e.g., class or field (cf. [Völ11]). Symbol kinds are an essential concept in the current thesis and impact the resolution process (cf. Chapter 6) as well as language composition (cf. Chapter 8).

Definition 3.2 (Symbol Kind)**.** *The* symbol kind *determines which information a symbol provides.*

Besides the information that can be associated with a symbol, the kind enables distinguishing same-named model elements of different kinds. As an example, a Java field and a Java method defined in the same class may have the same name whereas two same-named fields may not exist in the same class. For this, symbol kinds can be organized hierarchically.

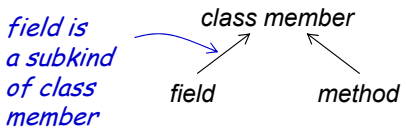


Figure 3.2: Symbol kind hierarchy by the example of class members.

Figure 3.2 highlights a simple symbol kind hierarchy (cf. Def. 3.2) by the example of Java class members. Fields and methods in Java are class members and have some information in common, e.g., they can be public and static. This information is summarized in the more abstract *class member* kind. Further, each has its specific information leading to the kinds *field* and *method* which both are of kind *class member*. Symbol kind hierarchies

are not restricted to a single language but can also be applied among different languages. That way, they increase efficiency of the language composition process (cf. Chapter 8).

Following the term "symbol table entry", Völkel [Völ11] refers to a symbol kind as "symbol table entry kind". Konat et al. introduce so-called *namespaces* for the "different kinds of names" [KKWV13] which is to some extend similar to our symbol kind concept. For instance, a class contains (i.e., "scopes" [KKWV13]) a dedicated namespace for methods and a dedicated namespace for fields. That way, methods and fields may be same-named. In contrast to a symbol kind, a namespace in [KKWV13] does not determine the information associated with the defined names. It rather is a scope (as in our terminology) for specific kinds of symbols. Finally, Parr [Par10] uses the term *category* which states "what kind of thing the symbol is".

Please note that both Def. 3.1 and Def. 3.2 do not make a statement about the abstraction level of a symbol or a symbol kind. The reason for this is that the abstraction level strongly depends on the purpose of a language (cf. essential model in Section 3.8), and thus, is a design decision of the language engineer [HMSNR15b]. For instance, dedicated symbols can represent the different Java types, namely class and interface (analogously enums). This follows from the fact that classes and interfaces in Java share many commonalities, e.g., consisting of fields and methods and extending supertypes. Furthermore, in most cases they cannot be distinguished in their usage, e.g., the type of a field can be a class or an interface without syntactical differences between them. This also applies for member accessing, e.g., `prop.getName()` does not (syntactically) indicate whether the type of `prop` is an interface or a class. Following from this, instead of defining two separate symbols, a single symbol can represent both classes and interfaces. Certainly, there are still differences between classes and interfaces. To name a few, an interface may not be declared as final and also cannot be instantiated.

Moreover, symbols can represent explicitly defined model elements as well as implicitly defined ones (cf. [GM10]). Symbols can also provide information that is not directly defined in the model element they denote but is somehow associated with that model element (cf. [HMSNR15b]). A symbol for a Java class, for example, can provide a transitive closure of all its visible methods, i.e., methods it defines itself and those it inherits from its supertype hierarchy, excluding overridden and private methods. Finally, a symbol can provide technical information concerning its represented model element, e.g., output-specific code generator information [MSNRR16].

3.4 Symbol References

A symbol definition can be referenced from different parts of the program. The local variable `sm` in Listing 3.1, for example, is *defined* in line 9 and *referred to* in lines 10 and 11. The referenced symbol does not need to be in the same model as, for instance, a superclass in Java, which usually is defined in another file.

Definition 3.3 (Symbol Reference)**.** *A* symbol reference *consists of the name and the kind of the used symbol that is defined elsewhere, e.g., in another scope [HMSNR15b].*

While there exists exactly one definition of a specific symbol, several references to that symbol may (co-)exist (cf. [Wat04, Che05]). Same as definitions, a symbol reference belongs to exactly one scope.

Depending on whether the referenced symbol is defined within the same model or in another model, we distinguish between two types of references, as defined in Def. 3.4.

Definition 3.4 (Intra-Model- and Inter-Model References)**.** *While* intra-model references *concern symbols within the same model* inter-model references *refer to symbols defined in other models.*

The term *reference* is used for the same concept in [Par10, KKWV13, NTVW15]. Some further terms are *use site* [KKWV13], and *applied occurrence* [Wat04, NTVW15]. Similarly, Völkel [Völ11] uses the term "reference" (in German "Referenz") (but realizes it differently, cf. Section 4.4.1). Throughout the current thesis, we use the terms "A refers to B", "A uses B", and "A references B" synonymously.

Besides symbol references consisting of the name and the kind of the referenced symbol—which forms the foundation for name-based model composition—, there also exist more complex references. In Java, for example, a generic type invocation additionally states one or more actual type arguments. This information is specific to the generic type invocation (or reference), not the definition.

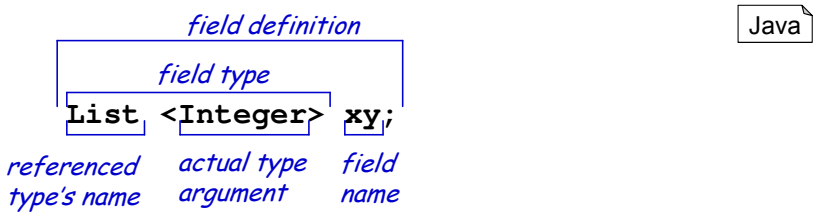


Figure 3.3: Distinction of symbol definition and reference by the example of a Java field.

Figure 3.3 highlights the difference by an example:

- The whole statement represents a field (or variable) *definition* named `xy`.
- The type of `xy` is a `List` of `Integers` (set via the generic type invocation `List<Integer>`). `List` is a *reference* to the same-named generic interface *defined* in package `java.util`.
- Further, the actual type argument is `Integer` (referring to `java.lang.Integer`). Obviously, this information is associated to the generic type invocation and not to `List`'s definition since other references can specify different type arguments, e.g., `List<String>` or `List<Boolean>`.

Please note that—in contrast to, for example, [GJS+14]—neither the angle brackets of generics (i.e., < and >) nor the squared brackets of arrays (i.e., [and]) are part of a reference's name, similar to [Völ11]. This is an essential aspect for the resolution process, which, among others, is based on the correct name (cf. Chapter 6).

3.5 Scopes

In block structured languages, model elements are defined in specific areas, which themselves can be nested. In the current thesis, we call those areas *scopes* (cf. [Par10, NTVW15]) while other typical names are *blocks* [GJR79, CWW80, ALSU06, GM10], or *namespaces* [Völ11]. We mainly use the term *block* to refer to the syntactical element (e.g., between curly brackets) which *spans* a scope. If a strict distinction is not necessary, we sometimes use "block" and "scope" interchangeably.

Definition 3.5 (Scopes)**.** *A* scope *holds a collection of symbol definitions (cf. [Par10]) and impacts their visibility (cf. Def. 3.6) [HMSNR15b, HLMSN+15a].*

In general, three forms of block structures can be distinguished (cf. [Wat04]):

- Languages with a *monolithic block structure* (e.g., older versions of Cobol) consist of only one block, leading to a single (global) scope, in which all symbols are defined. A major drawback is that symbol names must be unique throughout the whole program which, among others, impedes finding expressive names.
- An improvement was made by languages with *flat block structure*, such as Fortran, where the programs are "partitioned into several non-overlapping blocks" [Wat04]. Consequently, two kinds of variables exist; those that are global to the whole program and those that are local to the procedures in which they are defined.
- Finally, many modern languages have a *nested block structure*, where blocks can be nested within other blocks. This was first introduced in ALGOL-60 [BBG+63] and adapted in "ALGOL-like languages" [Wat04], such as C [KR88], Ada [Bar98], and Java [GJS+14]. In the current thesis, we are particularly interested in nested block structures since they are employed by many modern languages.

As mentioned above, a block (e.g., method block, if block, class block, etc.) spans a new scope (cf. [Sco09]). Consequently, scopes within a model are structured hierarchically [GJR79, CWW80, GM10] (in case of nested block structures), resulting in a tree-like scope graph[6] [CWW80, Seb08, HMSNR15b]. Thus, a scope has at most one direct *enclosing scope* (or outer scope [GJR79], surrounding scope [CL83], surrounding block

[6] While the scope structure within a model is a tree, the emerged (logical) scope structure among models (e.g., when considering superclasses) usually is a non-tree graph (cf. Section 3.10 and Chapter 6).

[ALSU06], exterior block [GM10], ancestor [CWW80], parent scope [NTVW15]) and several *subscopes* (or inner scopes [GJR79], interior blocks [GM10], subsequent scopes [GJR79, KKWV13], descendant [CWW80]). Since this thesis focuses on lexically scoped languages, the terms enclosing scope and *lexical enclosing scope* are used interchangeably (cf. *static parent* or *static ancestor* [Seb08]).

Please note that while a scope can have only one direct *lexical* enclosing scope it can have several *imported* scopes (e.g., scopes of supertypes in Java, cf. Section 3.10). Throughout this thesis, we always explicitly distinguish between these kinds of scopes. In contrast, for example, Parr [Par10] refers to both as *parent scope*, and further uses the term *enclosing scope* as in our terminology. Languages with monolithic scopes [Par10] consist of solely one scope, and hence, are nest-free [CWW80].

In the `System` class in Listing 3.1 (on page 31), the class scope defined in lines 3 to 16 encloses the method scope (lines 7–14), which itself encloses the if scope (lines 10–12).

Often, a scope (or its spanning block) is syntactically specified via a start and an end construct [Wat04, ALSU06, GM10, Par10], such as:

- `begin...end` (e.g., ALGOL-60 [BBG+63], Pascal [Wir71], Ada [Bar98])
- curly brackets `{...}` (e.g., C [KR88], Java [GJS+14], UML/P languages [Sch12, Rum16], MontiArc ADL [HRR12])
- round brackets `(...)` (e.g., Lisp [ABB+66])
- `let...in` (e.g., Haskell [HF92])
- `let...in...end` (e.g., ML [MTHM97])

3.5.1 Symbol Visibility

Definition 3.6 (Symbol Visibility)**.** *The* visibility *of a symbol is the logical region where the symbol is potentially accessible*[7] *by its (simple) name [HLMSN+15a, HMSNR15b] (cf. [Völ11, GJS+14]).*

The field `props`, for instance, is defined in the class scope (line 5, Listing 3.1), hence, it is visible within the whole class. In contrast, the local variable `sm` is defined in the method, and thus, cannot be accessed from outside the method where it is "out-of-scope".

The term "scope" is not used uniformly, sometimes even contradictorily. Occasionally, it is used as a synonym for visibility, e.g., "scope of a (name) declaration" (cf. [ALSU06, GM10, Völ11, GJS+14]). Clarke et al. use these terms in combination, e.g., "scope of an entity's visibility" [CWW80]. Scopes can also be considered from a reference's viewpoint

[7] Following Wolf et al. [WCW88], the definition says "potentially accessible" since shadowing (cf. Def. 3.7) may lead to a visibility hole.

as, for example, in [EB10, VS10]. That means, given a reference r and a (model) element e, the scope determines which elements of e are visible for r.

To eliminate name confusion, we use different terms for the area a symbol is defined in, i.e., its (enclosing) scope (cf. Def. 3.5) and the region in which a symbol is visible, i.e., its *visibility* (cf. Def. 3.6).

Symbols defined in a scope usually are visible in all their direct and indirect subscopes (cf. open scope [GJR79, CL83, GM10]). However, this does not necessarily hold true for subscopes defined in another model, e.g., in a subclass (cf. close scope [GJR79, CL83]). In such a case some kind of export and import mechanism is required. In Java, for example, a class *exports* all its non-private members which are *imported* in its subclasses via inheritance (cf. Section 3.10). Consequently, the visibility of a symbol is language-specific (although common approaches exist). It is determined by so-called *visibility rules* (or scope rules) [ALSU06, GM10].

3.5.2 Symbol Shadowing

Definition 3.7 (Symbol Shadowing)**.** *A symbol can be* shadowed *by symbols defined in importing scopes (usually subscopes) [HMSNR15b], and that way, its visibility is restricted. The* shadowing symbol *and the* shadowed symbol *typically are both same-named and of the same kind (or kind hierarchy).*

Some synonyms for *shadow* are *hide* [CWW80, Wat04, GM10, NTVW15], *redeclare* [CWW80, CL83], and *redefine* [GJR79].

Shadowing a symbol, leads to so-called *visibility holes* [GM10] in which the shadowed symbol is not visible. In Java, a local variable v in a method scope shadows a same-named field of the class scope. Thus, using the simple name v from *within the method* refers to the local variable, but from *outside the method* (and in the same class) it refers to the field. In contrast, since a local variable and a method have different kinds, neither can shadow the other.

A special case of shadowing (or hiding [GJS$^+$14]) is *method overriding* in languages such as Java and C#, where methods of superclasses are shadowed by methods that have the same signature, i.e, they are not only same-named but also have the same formal parameter declarations.

The Java specification [GJS$^+$14] explicitly distinguishes between shadowing and hiding of elements. While the former concerns only elements within the same class the latter references elements inherited from supertypes. We refer to both using the term “shadowing”. This is above all, because our generic scope graph does not require a strict distinction between shadowing and hiding (cf. Chapter 6).

3.5.3 Shadowing and Visibility Scopes

In some cases, not only the symbol kind and its name determine whether a symbol shadows another symbol but also the scope in which the (potentially) shadowing symbol is defined in. For example, the Java if block in Listing 3.1 cannot declare a new variable `sm` since its enclosing method already does. In contrast, the `getProperty` method may define `sm` even if a same-named field existed in the enclosing scope `System`. Therefore, we distinguish between two types of scopes:

Definition 3.8 (Shadowing and Visibility Scopes)**.** Shadowing scopes *(or symbols defined in them) may shadow symbols that are already defined in their imported scopes (e.g., enclosing scopes) whereas* visibility scopes *may not [HMSNR15b].*

Consequently, both the scope structure and the scope type impact the visibility of the containing symbols (cf. [CWW80, GM10, KKWV13]). In Java, class scopes as well as method scopes (and artifact scopes, cf. Section 3.5.5) are shadowing scopes. All other scopes (e.g., spanned by while, if, or for blocks) are visibility scopes.

Other approaches (e.g., [Par10, Völ11, KKWV13, Bet13, NTVW15]) do not introduce an explicit concept for shadowing scopes and visibility scopes. Instead, the shadowing ability is related to a symbol (e.g., a local variable) or to a concrete scope type (e.g., method scope). However, introducing the explicit concept of shadowing scopes (as in this thesis) simplifies to ensure that all symbols in a scope "behave uniformly with respect to name resolution" [NTVW15] since the scope's shadowing ability concerns all its contained symbols. In particular, this means, that (i) a symbol is either visible to all or none of the references in a specific scope, (ii) either all or none of the symbols in a scope can be referred to from outside of that scope, and (iii) every symbol in a scope can be referred to from any reference of that scope. Certainly, access modifiers as defined in Section 3.7 can also impact a symbol's visibility. For example, while a public field is visible from outside the class scope, a private field is not. However, none of the private fields is visible from outside, hence, those fields behave the same.

3.5.4 Named and Unnamed Scopes

Scopes defined by named model elements are named themselves, and thus, are called *named scopes* (e.g., [CL83, Par10, KKWV13]):

Definition 3.9 (Named and Unnamed Scopes)**.** *A scope, which has a name, is called* named scope*. Else, it is an* unnamed scope*.*

The class scope in Listing 3.1, for instance, is named `"System"`, same as the class. Named scopes impact the fully qualified name of a symbol (cf. Section 3.2). For example, the fully qualified name of the `props` field (line 5) is `java.lang.System.props` while `System` is the name of `props` enclosing scope. This also allows to distinguish the

`props` field of `System` from a same-named field of another class. Hence, named scopes typically are shadowing scopes.

In contrast—following from the observations of named scopes—, *unnamed* scopes (or *anonymous scopes* [KKWV13]) typically are visibility scopes, such as a scope spanned by an (Java or C#) if block. Although this holds true for many (modern) languages, it is not always the case. In C, for example, an if block spans a *shadowing scope*, and hence, allows to redefine names of its enclosing scope (cf. Section 6.2).

3.5.5 Artifact Scope

In textual languages, models are usually stored in an artifact, e.g., a file [Kra10, HMSNR15b]. In Java a top-level class is defined in a same-named *.java*-file. Analogously, models of the ADL MontiArc [HRR12] as well as models of the UML/P language family [Sch12, Rum16] are stored in respective files (e.g., *.arc*, *.cd*, etc.).

An artifact represents a separate compilation unit, and hence, "can be compiled on its own" [Wat04]. Following Krahn [Kra10], each artifact contains exactly one (public) model. Furthermore, an artifact optionally contains import and package information which concerns all model elements—and hence, the representing symbols—defined in that artifact. Therefore, those information are defined within the artifact scope.

Definition 3.10 (Artifact Scope (AS))**.** *The* artifact scope *represents the scope of the whole artifact (or compilation unit). It is the top scope of all symbols defined in an artifact and a shadowing scope [HMSNR15b].*

Some synonyms for the artifact scope are *file scope* [Che05, KKWV13] or *compilation unit scope* (cf. [Völ11]). The artifact scope plays an important role in the resolving process, especially for name qualifying and inter-model references, which are elucidated in Chapter 6.

Neron et al. [NTVW15] explicitly add a package scope to the scope graph and that way group all classes (or models) of the same package. While this is also possible with the approach presented in the current thesis, the package information is stated in the respective artifact scopes which reduces the depth of the scope graph. This also simplifies the implementation of the resolution process, as presented in Chapter 6.

Same as in our approach, Völkel [Völ11] does not provide an explicit package scope (namespace in his terminology). However, the package name is not stated in the compilation unit scope (see above) but is rather part of a symbol's (qualified) name.

3.5.6 Global Scope

Following Neron et al. that "[e]very program has at least one scope, the global or root scope" [NTVW15], we introduce the concept of the global scope (cf. [Che05, Par10]):

Definition 3.11 (Global Scope (GS))**.** *The* global scope *is the root scope of the whole scope graph incorporating all models. Furthermore, it maintains globally visible symbols including built-in predefined symbols.*

Please note that while the artifact scope is the top scope of a model, the global scope is the top scope *among all models*. The global scope's direct subscopes are artifact scopes. Further, it contains global types, such as `int` and `boolean` in Java. These types—which are neither defined in an artifact nor belong to a specific package—can be used in every model without being explicitly imported.

As for artifact scopes, both [Völ11] and [NTVW15] do not introduce an explicit concept of a global scope but rather allow to reuse the (generic) scopes for this task.

3.6 Scope Spanning Symbols

The name of a (named) scope (as introduced in Section 3.5) is specified by a named model element which in turn can be represented by a symbol (cf. Def. 3.1). Consequently, those model elements are represented by both a symbol and a scope. For this, we introduce the concept of scope spanning symbols:

Definition 3.12 (Scope Spanning Symbols)**.** *A symbol that represents a named model element which also spans*[8] *a scope, is called* scope spanning symbol *(cf. [HMSNR15b]).*

A symbol representing a Java class, for instance, spans a scope to enable field and method definitions within that scope.

Similarly, Neron et al. [NTVW15] use the term *module* to describe a scope spanning symbol. However, instead of making module a first-level concept, they specify three properties a "construct" (i.e., model element) must own in order to be classified as a module. These are [NTVW15]:

- having a name
- possessing an associated scope ("spanned scope" in our terminology)
- being importable into other scopes.

The properties listed above also apply to a scope spanning symbol as defined in Def. 3.12. The last bullet includes the implicit importing of an enclosing scope described in Section 3.10. A Java method, for example, is considered a scope spanning symbol, although it (or its local variables) cannot be (explicitly) imported into other scopes. However, it is (implicitly) imported in its contained subscopes (e.g., if block). Please

[8] or opens [CL83], introduces [ALSU06]

note that the symbol itself is only defined in its enclosing scope (cf. Section 3.5), not in its spanned scope.

While we explicitly distinguish between a symbol and its spanned scope, Parr uses the term *scoped symbol* to refer to a "symbol that also plays the role of a scope" [Par10]. However, separating these concepts allows for better reuse as demonstrated in Chapter 7. Völkel [Völ11] does not (technically) distinguish symbols that span a scope and those that do not. This results in some inconsistency issues between a symbol and its spanned scope, as discussed in Section 4.3.

3.7 Access Control Mechanisms

Many software languages provide mechanisms to control the accessibility [ALSU06] of model elements to enable encapsulation. In Java and C#, for example, the accessibility can be controlled via access modifiers, such as *public* (no restriction) and *private* (restricted to the current type), which sustain encapsulation [ALSU06]. Similar concepts exist for other languages, such as MontiArc and UML/P.

Definition 3.13 (Access Modifier)**.** *An* access modifier *restricts a symbol's visibility for outside use, i.e., via its qualified name (cf. [GJS+14]), and hence, determines its access from other symbols.*

Some access modifiers have an inclusion relation. For example, if a class may access *protected* members of another class, this implies that it can also access *public* members of that class. Consequently, the accessibility of the protected modifier includes the accessibility of the public modifier. In contrast, if a class may access public members, this does not include protected, package-local, or private members.

Definition 3.14 (Inclusion of Access Modifiers)**.** *Given two access modifiers m_1 and m_2, m_1* includes *m_2, if access rights to m_1 implies access rights to m_2.*

By this definition we infer that the inclusion relation is reflexive and transitive.

3.8 Symbol Tables

The concepts introduced in the previous sections are realized via a complex data structure that we call *symbol table* in this thesis. In the classical sense, a symbol table is a data structure (typically a hash table) that maps names to the associated information, i.e., it allows "to find the record for each name quickly and to store or retrieve data from that record quickly" [ALSU06]. For example, the name `props` in Listing 3.1 (line 5) on page 31 is associated with, among others, the information that it is a field with the type `Properties`.

In the current thesis, the symbol table is only conceptually a *table* but internally rather a *graph* of scopes (cf. Section 3.5). It still serves the goal of mapping a name to its associated information which is conducted via the scopes containing the symbols. That way, it also enables name-based linking of AST nodes (cf. [KRV10]).

Additionally, the symbol table in this thesis represents the essence, i.e., the *essential model* of a language [HMSNR15b]. According to Fowler [Fow10] an essential model (which he calls *semantic model*) "is a representation [...] of the same subject that the DSL describes" and is "[...] based on *what will be done* with the information from a DSL script". Please note that "what will be done" implies the language engineer's intention of how the language should be used. Thus, the essential model can contain information that is not (syntactically) stated in a model element but related to it, e.g., a collection of all inherited non-private fields (cf. Section 3.3).

Furthermore, the symbol table can omit irrelevant information to constitute an abstraction of the AST (cf. [NTVW15]), which includes (parts of) the language's interface. A *language interface* "is a relevant abstraction for a specific purpose of a provided or required part of a language [...]" [CvdBCR15]. The interface enables the symbol table user (see below) to focus on the relevant information of a model. In this regard, the symbol table shares some characteristics with the notion of a *model type* which "defines an interface to manipulate models" [DCB+15] as employed in the Melange framework (based on [SJ07]). However, unlike model types in Melange, the symbol table does not primarily aim at enabling substitutability of models (cf. [GCD+12]), although the kind hierarchy introduced in Section 3.3 enables this to some extend (cf. Chapter 8). Moreover, a model type is a structural interface "over the abstract syntax of a language" [DCB+15]. The symbol table in this thesis is part of a language's abstract syntax, which describes "essential concepts and structure of the sentences without semantically irrelevant concrete sugar" [CvdBCR15].

In sum, a symbol table in the current thesis is defined as in Def. 3.15 (cf. [ALSU06, Völ11, HLMSN+15a, HMSNR15b]).

Definition 3.15 (Symbol Table (ST))**.**

1. *The ST is a data structure consisting of a scope graph with an associated collection of symbols in each scope*[9]*. It* maps names to essential information *about model elements, represented as symbols. The ST allows to efficiently organize and find, among others, declarations, types, and implementation details associated with those model elements.*

2. *Moreover, the ST enables to efficiently navigate between AST nodes of the considered model(s).*

[9] Following from this, we sometimes refer to a model's scope graph and the contained symbols as the *symbol table of that model*.

3. *Furthermore, the ST represents the* essence of a language*, i.e., of its models. This especially includes the* model interfaces *constituted by the language interface (cf. [CvdBCR15]).*

Roles

The language engineer (or language developer [KRV06]), among others, defines the syntax of a language. A language user employs the (abstract) syntax of the language, e.g., for code generation (cf. tool developer [KRV06]). Since this thesis focuses on symbol tables—which are a part of the abstract syntax—we sometimes write *symbol table engineer* and *symbol table user* to refer to a language engineer and a language user, respectively.

3.9 Symbol Table Notation

This section presents a graphical notation for a better illustration of the main symbol table elements introduced throughout this chapter, i.e., the emerging scope graph with the different types of scopes, and the contained symbol definitions and symbol references including the respective symbol kinds. Same as an object diagram, it represents an instance of a symbol table but allows a clearer and more intuitive representation of the underlying concepts. Figure 3.4 gives an overview of the graphical elements. The Java listings only serve for a better understanding of the scope graph and are not further considered.

Circles represent scopes, e.g., *D* and *if*. While the former is a shadowing scope (solid line), the latter is a visibility scope (dashed line). The circles labeled with *GS* and *AS* embody the global scope and artifact scopes, respectively. Italicized circle names are used for unnamed scopes, e.g., the *if* scope. The enclosing-sub relation is illustrated by lines between the scopes, whereas the visually upper scope depicts the enclosing scope. The relation between the global scope and the artifact scopes is depicted by two lines to emphasize that (i) the relation is outside the artifact and that (ii) the resolving request is multiplied when leaving the artifact since, among others, import statements are considered as well. This process is discussed in detail in Chapter 6.

Symbol definitions and references are presented by rectangles and rounded shapes, respectively. Additionally, an optional icon represents the symbol kind, and hence, symbols with the same icon have the same kind (analogously for symbol references). We will omit the symbol kind in most examples if an explicit distinction is not necessary.

Lastly, scope spanning symbols can be represented by either circles with a contained rectangle (e.g., *m*), or a same-named scope and symbol at the same scope hierarchy (optionally connected by an arrow), e.g., scope *D* and symbol *D*. In general, we will omit parts of the scope graph that are not important for the topic under discussion, highlighted by `...` (three dots) below the *ST* flag at the top on the right.

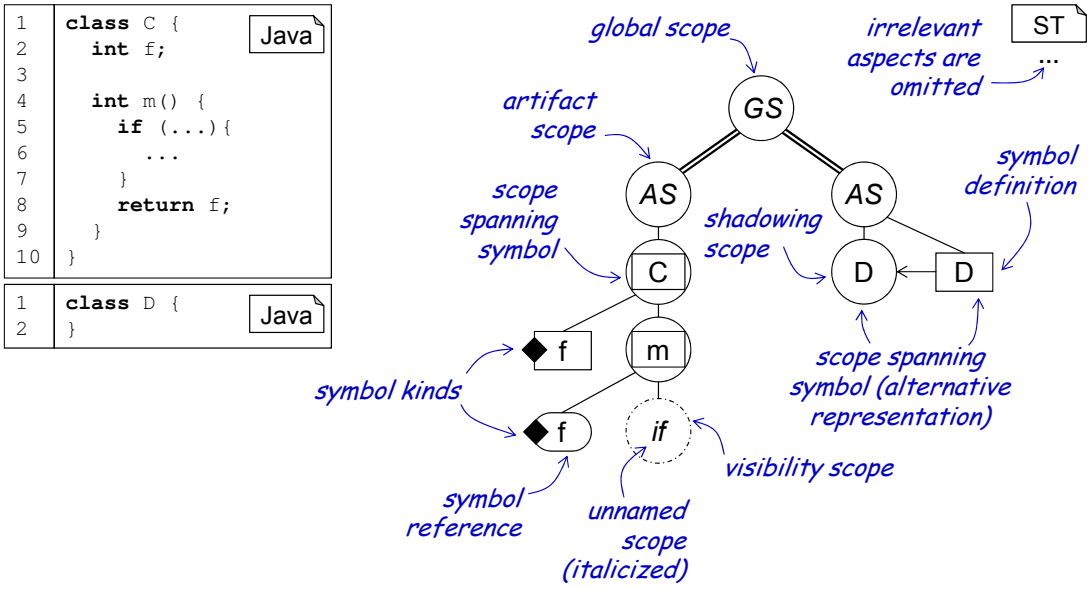


Figure 3.4: Graphical notation for symbol table elements.

Neron et al. [NTVW15] also use a graphical notation for representing their scope graph. In contrast to our approach, they mainly aim at depicting possible (name) paths in the scope graph. This leads to some essential differences compared to our approach, although some notational elements are the same (e.g., circles and squares):

- Symbol kinds do not exist in the scope graph of [NTVW15].
- Scopes in [NTVW15] are numbered depending on their occurrences order (not named as in our approach).
- The different positions of names and their references are illustrated in [NTVW15]. That way, a scope in the scope graph sometimes contains the same name several times using indices like x_i, where i is the corresponding position.
- There is no (visual) distinction between shadowing and visibility scopes.
- There is no (visual) distinction between named and unnamed scopes.
- No dedicated notations for a global scope or artifact scopes exist.
- While the scope graph in [NTVW15] is directed—which is important for finding a valid path—, ours is not. The reason will be clarified in Chapter 6.

3.10 Encapsulated, Exported, Imported, and Forwarded Symbols

The (lexical) scope hierarchy (cf. Section 3.5) as well as access modifiers (cf. Section 3.7) lead to so-called encapsulated, imported, exported, and forwarded symbols [HLMSN+15a] (cf. symbol table kinds in [Völ11]). In the following, these are introduced by the example of the class structure presented in Figure 3.5.

The top part of Figure 3.5 shows a class hierarchy with three classes. The emerged (logical) scope structure is depicted in the bottom part. Class *A* has the two fields *a1* and *a2*. Since *a1* is private, it is only visible within *A*'s spanned scope (and its subscopes, if existed). Hence, *a1* is *encapsulated* in scope *A* (cf. "local variable", [Wat04]).

Definition 3.16 (Encapsulated Symbol)**.** *An* encapsulated symbol *is only visible within its enclosing scope and its (lexical) subscopes (if not shadowed by other symbols).*

The field *a2* is public, and thus, can also be referenced from outside, e.g., by class *B*. Hence, *A exports a2* for outer usage.

Definition 3.17 (Exported Symbol)**.** *An* exported symbol *can be used from outside its enclosing scope, among others, depending on its access modifier.*

The class *B* subclasses *A*. That way, it can additionally access the exported symbol *a2* (but not the encapsulated *a1*). Consequently, two symbols are visible within *B*'s scope:

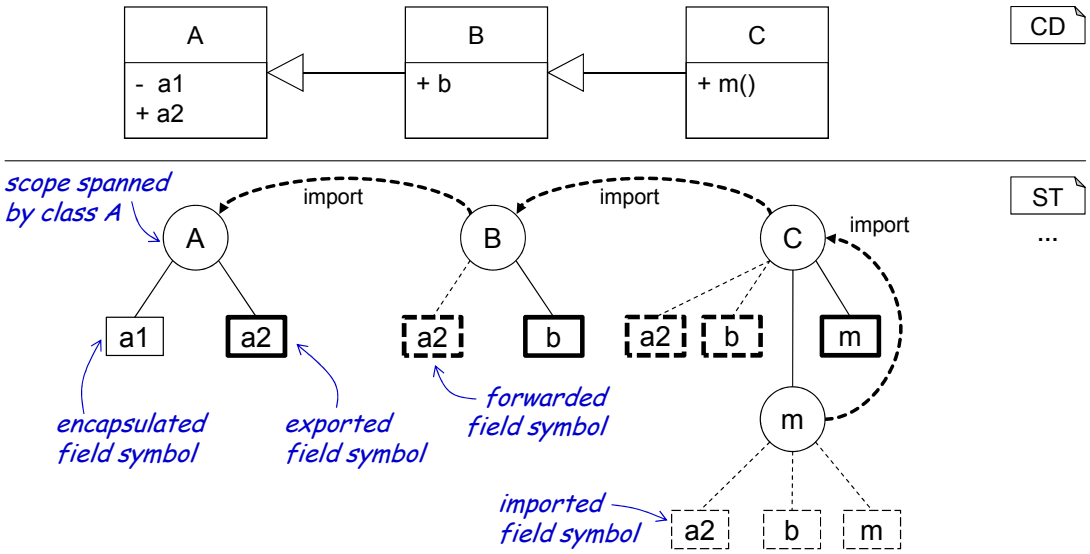


Figure 3.5: Encapsulated, exported, imported, and forwarded symbols by the example of a class hierarchy.

field *b*, which is directly (i.e., lexically) defined in *B*, and field *a2* (marked by dashed lines) defined in class *A*. Unlike *b*, *a2* is not stored in *B*'s scope but only visible in it, following Def. 3.1 that a symbol is defined exactly once. *a2* is said to be *imported* into *B*.

Definition 3.18 (Imported Symbol)**.** *Scopes can* import *symbols that are exported (cf. Def. 3.17) by other scopes. Symbols of lexical enclosing scopes are automatically imported, even if they are not (explicitly) exported (cf. open scope [GJR79, CL83]).*

Based on this definition, we sometimes write "scope *S* imports scope *T*" throughout the thesis, which is an abbreviation for "scope *S* imports all (visible) exported symbols of scope *T*". Consequently, *T* is an *imported scope* of *S*. Sometimes, *inherit* is used as synonym for *import* (e.g., [GJR79, GM10]).

In Java a class inherits all (visible) methods and fields of its class hierarchy, hence, *b* as well as *a2* (in Figure 3.5) are visible in class *C*. The latter is visible since *B* not only imports *a2*, but also exports it. Consequently, *B forwards a2*.

Definition 3.19 (Forwarded Symbol)**.** *A symbol that is both imported and exported is called a* forwarded symbol.

Finally, *C*'s spanned scope defines method *m* and imports (or forwards) *b* and *a2*. Since the scope of method *m* is a subscope of *C*'s scope, it (automatically) imports all visible symbols of *C*, i.e., *m*, *b*, and *a2*. That way, the symbols exported by *A* and *B* are *transitively* imported in *m*.

Please note that the import relation between *m* and *C* exists *implicitly* via the enclosing-sub relation of these scopes (cf. open scope [GJR79, CL83]). In other words, since *C* is the *lexical* enclosing scope, *m* imports (or inherits [GJR79]) all of its symbols including private ones. In contrast, the import relation between *C* and *B* (analogously *B* and *A*) is *explicitly* stated via the *extend* relation. Furthermore, a scope usually imports all (non-shadowed) symbols of its enclosing scope even if they are not exported, whereas a class scope only imports the exported (or forwarded) symbols of its superclass' scope (cf. open and closed scopes [GJR79, CL83]).

Importing a scope is not the same as importing a type with, for example, the Java import statement which is a *private import*. Figure 3.6 highlights the difference. In contrast to Figure 3.5, class *B* does not *extend A* but *imports* it. Consequently, *B*'s scope only imports the symbol representing *A* (not its field *a*), as shown in the bottom part of Figure 3.6. Unlike the previous case, the imported symbol is not exported, and hence, is not a forwarded symbol. Consequently, *A* is not visible within *C*.

Please note that both Figure 3.5 and Figure 3.6 well illustrate the *essence* (cf. Def. 3.15) of the depicted classes and their relations, that is, among others:

- How scopes within the same model as well as between different models are related.
- Which symbols are visible in which scopes (although not lexically defined there).

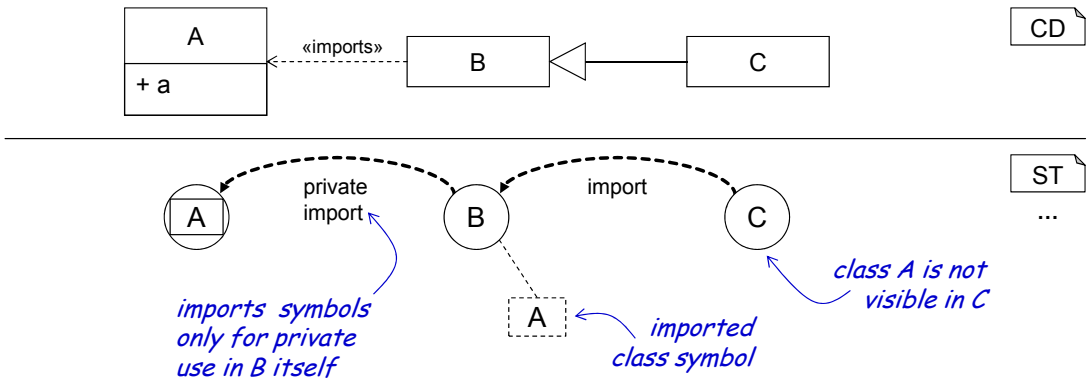


Figure 3.6: Example of privately imported symbols.

Conceptually, access modifiers lead to different interfaces of a model (cf. [Völ11]), e.g., an interface consisting of only public members, and an interface that additionally provides protected members. Völkel [Völ11] realizes those model interface kinds via separated symbol table kinds:

Encapsulated symbol tables only contain entries (symbols in our terminology) that are visible within the enclosing namespace (scope in our terminology). This is quite similar to our approach.

Imported symbol tables contain entries that are imported from other namespaces (and not shadowed in the current namespace). The main difference to our approach is that for each access modifier (e.g., public and protected) Völkel provides a dedicated imported symbol table. In the current thesis, the different kinds of access modifiers are explicit concepts belonging to a symbol.

Exported symbol tables contain all entries that are exported, i.e., visible from outside the namespace. Same as before, for each access modifier a dedicated symbol table exists. Again, this thesis provides top-level concepts for access modifiers instead.

Forwarded symbol tables contain entries that are both imported and exported. For example, an inherited non-private Java method is imported from the namespace of the superclass and also exported, and hence, can be imported by subclasses.

Same as [Par10, VS10, Bet13, KKWV13, NTVW15] we do not introduce additional containers (e.g., scopes) for separating symbols with different accessibility. Instead, access modifiers are a top-level concept in the current thesis and explicitly considered during the resolution process. The advantages are discussed in Chapter 6.

3.11 Method for Finding Candidates for Symbols and Scopes

As described in Section 3.3, a symbol represents a named model element and its essential information. Figure 3.7 and the following questions help the language engineer deciding for which model elements corresponding symbols and/or scopes can be useful.

If the model element has a name, very likely a symbol is required that represents essential information of that model element. The reason is that in many textual languages names are used to refer to model elements. Expressive names simplify understanding the meaning of a model (or model element). Moreover, names can be part of a model's interface [Rum13]. If a named model element itself contains named model elements, it is a candidate for a scope spanning symbol.

After finding candidates for symbols, it has to be determined which ones finally result in a symbol. For that, the following questions can help:

Q1 Is the model element referred to from within the model by other model elements?

Q2 Can it be referenced by other models?

Q3 Does the model element embody essential information? For example, is it part of the model's interface (cf. Section 3.8)?

Q4 Is the model element affected by visibility rules such as shadowing (cf. Def. 3.7)?

If (at least) one of the above questions is answered with "yes", the model element should be represented by a symbol (or scope spanning symbol). It is also sufficient if a question is affirmed depending on the context of the model element. Object-oriented

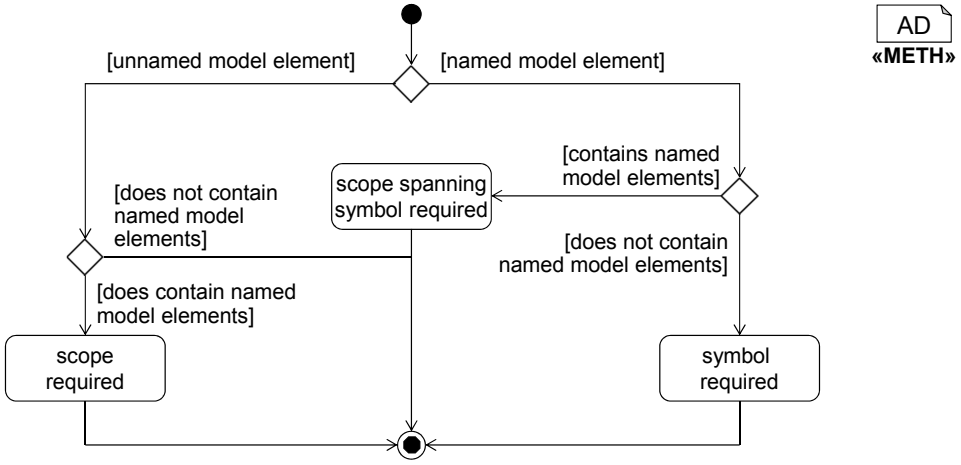


Figure 3.7: Method for determining candidates for symbols and scopes.

languages, for example, provide access modifiers to set the visibility of a method. Hence, whether a method can be referenced by other models (question **Q2**), depends on its access modifier. Nevertheless, a method should be represented by a symbol (even if the other questions would not apply). This also simplifies the context condition checks which can make use of the symbol table.

To determine whether a symbol spans a scope, and hence, is a scope spanning symbol, the following question must be answered with "yes":

Q5 Does the model element contain model elements that are represented by a symbol?

Question **Q5** also helps to find candidates for scopes; an unnamed model element that contains named model elements (cf. Figure 3.7). Furthermore, some syntactic elements, such as brackets (cf. Section 3.5), can help to identify candidates for scopes.

Figure 3.7 focuses on determining whether a specific model element should be represented in the symbol table. The decision is conducted sequentially for each model element. For this, an approach as depicted in Figure 3.8 can ease the decision process, which is divided into three phases. In the first phase, symbols are specified by iterating through all *named* model elements (questions **Q1** - **Q4**). In the second phase, it is determined which of those symbols is a scope spanning symbol (question **Q5**). Finally, for the remaining model elements it is determined, which of those elements spans a scope. Since in the previous two phases symbols are identified, in this phase solely model elements have to be considered which contain model elements that are represented by symbols (question **Q5**).

[ZH11] present a method for finding the essence of structural models based on UML class diagrams. The method can be applied to the AST classes (generated from MontiCore's grammar), to help determination of an appropriate symbol table structure (as a language's essential model, cf. Def. 3.15).

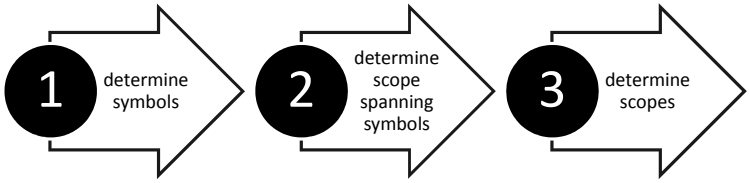


Figure 3.8: Determining symbols and scopes in three phases.

Chapter 4

SMI: Symbol Management Infrastructure

This chapter presents the symbol management infrastructure *SMI*, which has been developed as part of the MontiCore 4 runtime[1] in the context of this thesis. SMI serves as (name and kind based) *integration structure* and *manager for models* (and model parts) of the same language (cf. Chapter 6) as well as heterogeneous languages (cf. Chapter 8). Same as the features and concepts discussed in Chapter 3, SMI results from experiences gained from many tools and languages developed with Völkel's infrastructure [Völ11] (on top of previous MontiCore versions [Kra10]), such as the UML/P languages [Sch12, Rum16], JavaScript and TypeScript languages for MontiCore [Nes13], the RoboTask language family [HMSNR+15a], software categories languages [NN14, MSNR15a, MSNR15b], as well as MontiArc [HRR12, www16i] and its extensions MontiArc^{HV} [HRR+11], Δ-MontiArc [HRRS11, HKR+11], MontiArcAutomaton [RRW12, RRW13a, RRW13b, RRW14], cloudADL [NPR13], and MontiSecArc [HHRW15].

SMI provides a technical realization of many concepts introduced in Chapter 3, namely, symbol definitions and their kinds (cf. Section 3.3), symbol references (cf. Section 3.4), scopes (including the different scope types, cf. Section 3.5), scope spanning symbols (cf. Section 3.6), and access modifiers (cf. Section 3.7). Figure 4.1 depicts an overview of the main types involved, leaving out technical details. As it can be seen, SMI provides a designated type for each of the above mentioned concepts. In addition, a default class (not shown in Figure 4.1) exists for most of these types, having the same name as the interfaces with the prefix `"Common"`, such as `CommonScope` and `CommonSymbolReference`.

Moreover, this chapter elaborates many patterns which *serve as methods* for developing language-specific symbol tables (cf. Section 3.8). Sometimes, the chosen pattern can determine whether a language-specific symbol table can be (efficiently) composed or not (cf. Chapter 8). The pattern catalog is based on (i) lessons learned from the above listed tools developed with previous MontiCore versions as well as (i) experiences gained from tools developed with SMI and MontiCore 4, such as NESTML [PBI+16], MontiArc (migrated to MontiCore 4), MontiJava [Mul15], OCL [Cel15], JavaScript [Sie15], CD4Analysis (a restricted UML/P class diagram language [Sch12, Rum16]), and object-diagrams.

[1] see `https://github.com/MontiCore/monticore/tree/master/monticore-runtime`

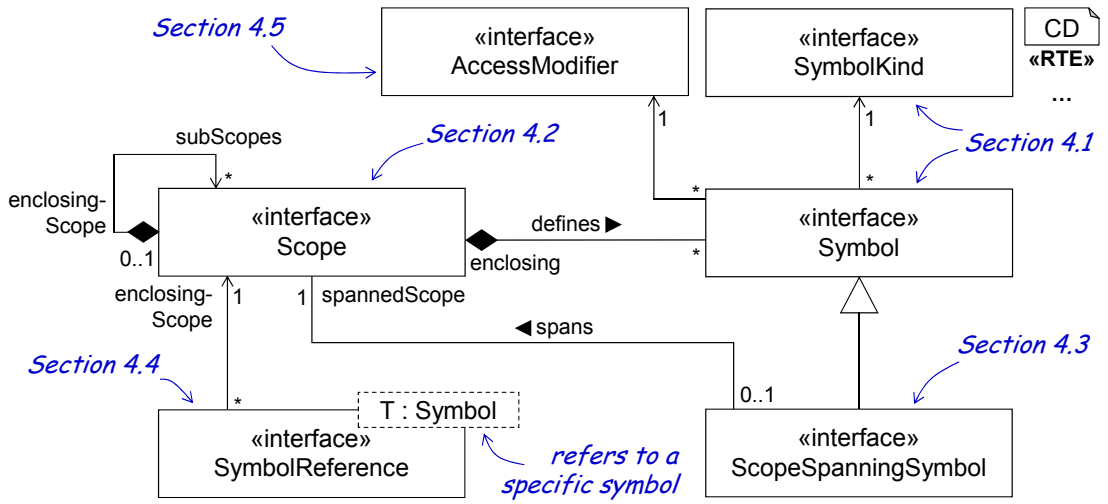


Figure 4.1: Overview of the main technical interfaces for the concepts introduced in Chapter 3. SMI provides default implementations for each of these interfaces.

This chapter further discusses some design decisions made in SMI not only for enabling efficient and effective development of language-specific symbol tables but also with respect to generating them (cf. Chapter 7) as well as composing them with symbol tables of other languages (cf. Chapter 8).

Since much related work has already been discussed in Chapter 3, the current chapter mainly focuses on the *technical aspects* of other works. Language workbenches such as Xtext [Bet13] and EMFText [HJK+09], for example, are based on the EMF framework [SBPM09], and hence, rely on its generic infrastructure which sometimes might be too heavy-weighted for the given task. In contrast, SMI is tailored to the symbol table domain, and hence, omits aspects not related to it. Parr [Par10] gives a good overview about how to implement a language application. Although at a first glance our presented infrastructure is similar to [Par10], it—quite the opposite—is different in many ways. This is, among others, because Parr focuses on developing single languages while the current thesis also aims at composing languages, even a-posteriori (cf. [HLMSN+15a, HLMSN+15b]). The Spoofax language workbench [KV10, www16n] conducts the semantic analysis using the Stratego program transformation language [Vis01, BKVV08]. For this, the AST is, among others, "desugared" and "decorated" [KV10]. While the former simplifies the AST the latter adds additional semantic information to it. The resulting AST can be considered as the symbol table structure (in combination with the AST) being the essential model, as presented in the current thesis. However, since Spoofax employs meta-languages for (declaratively) specifying languages, in contrast to our Java-based approach, its underlying technical concepts differ fundamentally from those presented throughout this chapter.

Chapter Outline

The Sections 4.1, 4.2, 4.3, 4.4, and 4.5 present technical classes for symbols (cf. Def. 3.1) and their kinds (cf. Def. 3.2), scopes (cf. Def. 3.5), scope spanning symbols (cf. Def. 3.12), symbol references (cf. Def. 3.3), and access modifiers (cf. Def. 3.13), respectively. Additionally, each of these sections suggests and discusses several patterns for language-specific implementations of the presented classes. Next, Section 4.6 introduces the technical realization of the relation between AST nodes and symbol table elements. Finally, Section 4.7 suggests some naming conventions for language-specific classes.

4.1 Technical Realization of Symbols and Symbol Kinds

SMI provides dedicated interfaces for symbols (cf. Def. 3.1) as well as symbol kinds (cf. Def. 3.2). As depicted in Figure 4.2, `Symbol`'s default implementation is provided by the abstract class `CommonSymbol` (see below) while `SymbolKind` provides its default implementation via the `default` modifier newly introduced in Java 8 [GJS+14]. This is only possible in case the implementation is stateless. Each specific symbol kind must (directly or indirectly) subtype `SymbolKind`.

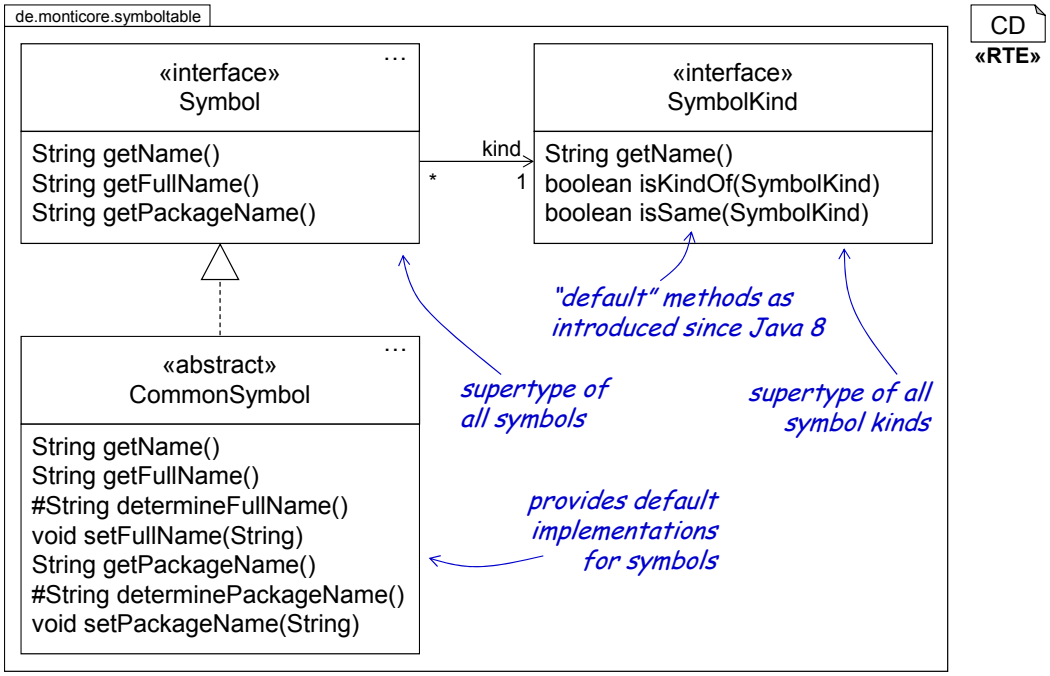


Figure 4.2: Overview of the `Symbol` interface, its default implementation `CommonSymbol`, and the `SymbolKind` interface.

```java
1 public interface SymbolKind {
2
3   default String getName() {
4     return SymbolKind.class.getName();
5   }
6
7   default boolean isKindOf(SymbolKind kind) {
8     return kind.getName().equals(this.getName());
9   }
10
11  default boolean isSame(SymbolKind kind) {
12    return this.isKindOf(kind) && kind.isKindOf(this);
13  }
14 }
```

Java «RTE»

Listing 4.3: Default implementations for the methods `getName`, `isKindOf`, and `isSame` of the `SymbolKind` interface.

Listing 4.3 presents the default implementation of `SymbolKind`'s methods:

`getName()` Returns the name of the symbol kind (lines 3–5, Listing 4.3) which is required in the `isKindOf` method. In order to reduce potential name clashes, a unique name should be used as, for example, the qualified name of the symbol kind class (line 4, Listing 4.3).

`isKindOf(SymbolKind)` Enables a symbol kind hierarchy, as described in Section 3.3. The method checks whether a symbol kind `k1` has the kind of a symbol kind `k2`. By default, this is true, if `k1` and `k2` are same-named (line 8, Listing 4.3).

Listing 4.4 demonstrates an implementation of `isKindOf` for language-specific kinds. The name comparison (line 2, Listing 4.4) ensures that a symbol kind is a kind of itself. Additionally, the `isKindOf` method of the direct supertype (here `SymbolKind`) is invoked (line 3), which enables the kind hierarchy.

To avoid that each language-specific symbol kind class has to override the `isKindOf` method, Java's Reflection API[2] can be exploited. As shown in Listing 4.5, with reflection the default implementation of `isKindOf` can be implemented once and does not need to be overridden by subtypes. This, however, requires the symbol kind classes to be in a (technical) class hierarchy.

`isSame(SymbolKind)` Checks whether two symbol kinds `k1` and `k2` represent the *same* kind, i.e., `k1.isKindOf(k2)` and `k2.isKindOf(k1)` (line 12, Listing 4.3).

[2] see `https://docs.oracle.com/javase/tutorial/reflect/`

```java
public boolean isKindOf(SymbolKind kind) {
  return kind.getName().equals(this.getName())
    || SymbolKind.super.isKindOf(kind);
}
```

Java «LS»

Listing 4.4: Implementation of `SymbolKind`'s `isKindOf` method for language-specific symbol kinds.

```java
  public boolean isKindOf(SymbolKind kind) {
    return
      kind.getClass().isAssignableFrom(this.getClass());
  }
```

Java «RTE»

Listing 4.5: Implementation of `SymbolKind`'s `isKindOf` method via reflection.

Similar to `SymbolKind`, the `Symbol` interface depicted in Figure 4.2 is the superinterface of all types representing a symbol (cf. Def. 3.1). Figure 4.2 shows some of the methods provided by `Symbol`. The methods concern the different sorts of a symbol's name (cf. Section 3.2), i.e., the simple (or unqualified) name and the (fully) qualified name. For these, the `Symbol` interface provides the methods `getName` and `getFullName`, respectively, and also the method `getPackageName` to obtain the package name:

`getName()` Returns the symbol's unqualified name, e.g., `"List"`.

`getFullName()` Returns the symbol's fully qualified name, e.g., `"java.util.List"`.

`getPackageName()` Returns the symbol's package name, e.g., `"java.util"`.

While the simple name is directly stated in a symbol s (e.g., `"List"` in `interface List`), the fully qualified name q depends on the context of s, i.e., its enclosing scope hierarchy, (cf. Section 3.2), and is computed as follows:

1. if q is set (i.e., it is not `null`), stop, else continue with next step
2. set $q := s.name$
3. traverse up the enclosing scope(s) e_i (for $i \in \{0, 1, 2, ...\}$) of s beginning with its direct enclosing scope e_0, i.e., $i = 0$
   a) if e_i is spanned by a symbol t, set $q := t.fullName + "." + q$, where $+$ is string concatenation, and stop (t's full name is reused).
   b) else if e_i is a *named* scope (cf. Def. 3.9), set $q := e_i.name + "." + q$.
   c) else, i.e., e_i is unnamed, stop.
4. finally, if s has a package definition p, set $q := p + "." + q$.

```
1 public String getFullName() {
2   if (fullName == null) {
3     fullName = determineFullName();
4   }
5
6   return fullName;
7 }
```

Java «RTE»

Listing 4.6: The `getFullName` method of `CommonSymbol` class.

The two methods `getFullName` and `determineFullName` of the `CommonSymbol` class provide default implementations for retrieving the full name. The former solely conducts the first check (cf. Listing 4.6) and then delegates to the latter, which then conducts the steps *2.–4.*

As it can be seen in Listing 4.6, the full name is only determined (line 3) if it is not already set (line 2), e.g., via `setFullName`. If the language engineer needs to adjust the default behavior, she only has to override the `determineFullName` method which serves as hook method [Pre95a]. Alternatively, she can ignore the automatic computation and solely use `getFullName` and `setFullName` as usual accessor and mutator methods, respectively (ensured by steps *1* and *3a*). Both `getFullName` and `determineFullName` concern a symbol's *definition*, i.e., all required information is available. Hence, these methods always succeed.

A symbol's fully qualified name plays a central role in the resolution process (cf. Chapter 6). Hence, when customizing the name, it must be ensured that the name is still consistent with its defining scope hierarchy. For example, a symbol m defined in a scope e may not have a qualified name $x.m$ which does not match its enclosing scope's name. Instead, its qualified name must be $e.m$ (*step 3b*).

Determining the full name of a symbol depending on its context (i.e., the scope it is defined in) yields some benefits. Firstly, it is less error-prone (e.g., symbol's qualified name must match its enclosing scope's name) and eases the work of the language engineer since she does not need to set the full name of each symbol manually. Secondly, when embedding languages (cf. Section 8.2), the qualified names of embedded model elements are automatically derived, e.g., a state in a statechart might have a different fully qualified name than a state embedded in a class.

However, there are two limitations. First, overloaded symbols are *same-named*. For example, overloaded Java methods have the same name but differ in their parameters. Consequently, the qualified name of the `getProperty(String)` method and the overloaded method `getProperty(String, String)` of the class `java.lang.System` both have the qualified name `java.lang.System.getProperty`[3]. To handle those

[3]The Java language specification [GJS+13] states that a "qualified method name can only appear in the context of a method invocation expression". Since such an expression includes the argument types, the corresponding method can be resolved uniquely.

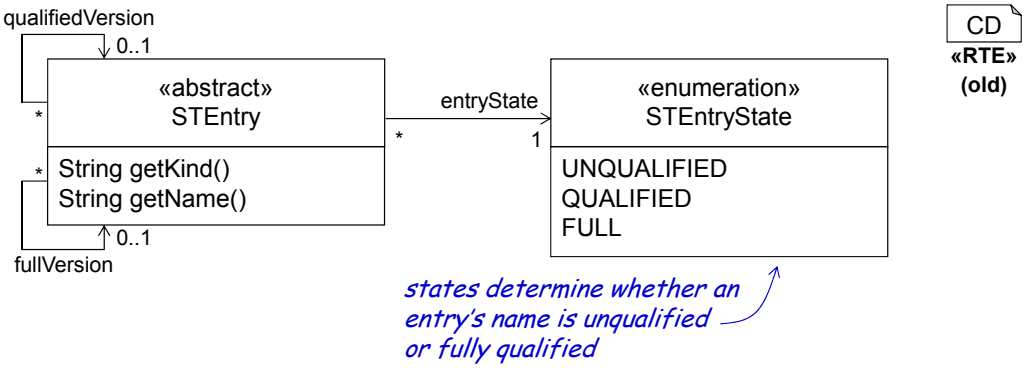


Figure 4.7: Symbol table entry states as suggested by Völkel [Völ11]. The states, among others, determine whether `getName` returns an unqualified name (state `UNQUALIFIED`) or a fully qualified name (either state `QUALIFIED` or `FULL`).

cases, the symbol resolution mechanism of SMI (cf. Chapter 6) allows to pass further information (such as formal parameter types) in order to identify a model element. The second limitation when calculating a symbol's fully qualified name occurs when one of its enclosing scopes is unnamed (cf. Def. 3.9), such as an if block in Java. Such scopes cannot be located by their name, hence, the contained model elements cannot be referenced. In those cases, the determination of the full name stops (step *3c*), and is same as the simple name (or partially qualified name). This limitation, however, is rather a feature in most languages where elements in unnamed blocks cannot be referenced (cf. Chapter 6). Although it is possible to specify artificial names like *if1*, *if2*, etc., this is not recommended since it can lead to undesired behavior which is hard to track especially when the model changes.

The package name of a symbol—whether top-level or not—is considered to be the package as declared in its enclosing artifact (cf. Section 3.5.5). For instance, the package of the (top-level) class `java.lang.System` and its defined fields is the same, namely `"java.lang"` as declared in the artifact scope. Analogously to the full name, the package name of a symbol is calculated automatically.

Figure 4.7 presents the structure of a symbol table entry (called "symbol" in our terminology) as suggested by Völkel [Völ11]. Same as the `Symbol` interface provided by SMI, `STEntry` represents the abstract supertype of all symbol table entries. There are, however, two essential differences to our approach. First, Völkel does not provide a dedicated type for symbol table entry kinds, such as `SymbolKind` in the current thesis. Instead, the kind is a string stated by the `getKind` method. Although this simplifies the class structure, it does not allow for (at least not with few effort) defining kind hierarchies. For example, its not possible to state that a Java class and a C# class both are of kind `"class"` and additionally have their own specific kinds `"java.class"`

and "csharp.class", respectively. Providing a dedicated type SymbolKind, SMI nullifies these disadvantages (cf. Section 4.1). The second difference is that a symbol table entry in [Völ11] has up to three states (cf. Figure 4.7) which, among others, determine whether the name is unqualified (i.e., UNQUALIFIED) or (fully) qualified (i.e., QUALIFIED or FULL). Unlike Symbol in our approach, STEntry provides only one method for retrieving the name. Depending on the entry's state, getName returns the unqualified or qualified name of the entry.

Similar to SMI, the Symbol class in [Par10], among others, provides the information about a symbol's name and its enclosing scope. In contrast, it does not explicitly distinguish between the different forms of names. The most essential difference, however, is that symbol kinds do not exist explicitly (e.g., as interfaces) in the framework. Hence, there is neither a (generic) way to search for symbols of specific kinds nor is it possible to define kind hierarchies, as in the current thesis. The kind hierarchy in [Par10] is rather an implicit part of the (technical) class hierarchy.

In EMF-based frameworks, such as Xtext [Bet13] and EMFText [HJK+09], the interface ENamedElement represents a named model element, and thus, is comparable with the Symbol interface of the current thesis. However, ENamedElement solely provides the simple name of the model. In order to determine the qualified name usually specific providers exist (cf. Section 6.11).

MPS [VS10] follows a projectional approach which means that the AST is the core structure specified by so-called *concepts*. Moreover, named concepts exist, similar to symbols in the current thesis (which represent named model elements). Concepts also have the role of symbol kinds as in our approach, and allow checks such as c.isSubConceptOf(Field) and c.isExactly(Field) similar to the methods provided by SymbolKind.

After introducing SMI's technical classes for symbols and symbol kinds, the remainder of this section suggests some patterns for realizing language-specific symbols and their kinds and further discusses advantages as well as disadvantages of the patterns.

4.1.1 Patterns for Redundant Information Contained in a Symbol and its Related AST Node

If a symbol is created from an AST node (which is typically the case, cf. Chapter 5), one of the three cases occurs (cf. Figure 4.8): (i) the symbol does not provide any information that its related AST already contains, (ii) the symbol provides the information of its AST node via delegation, (iii) the symbol provides the information of its AST node without delegation, and thus, has some redundancy. The following presents patterns for these cases.

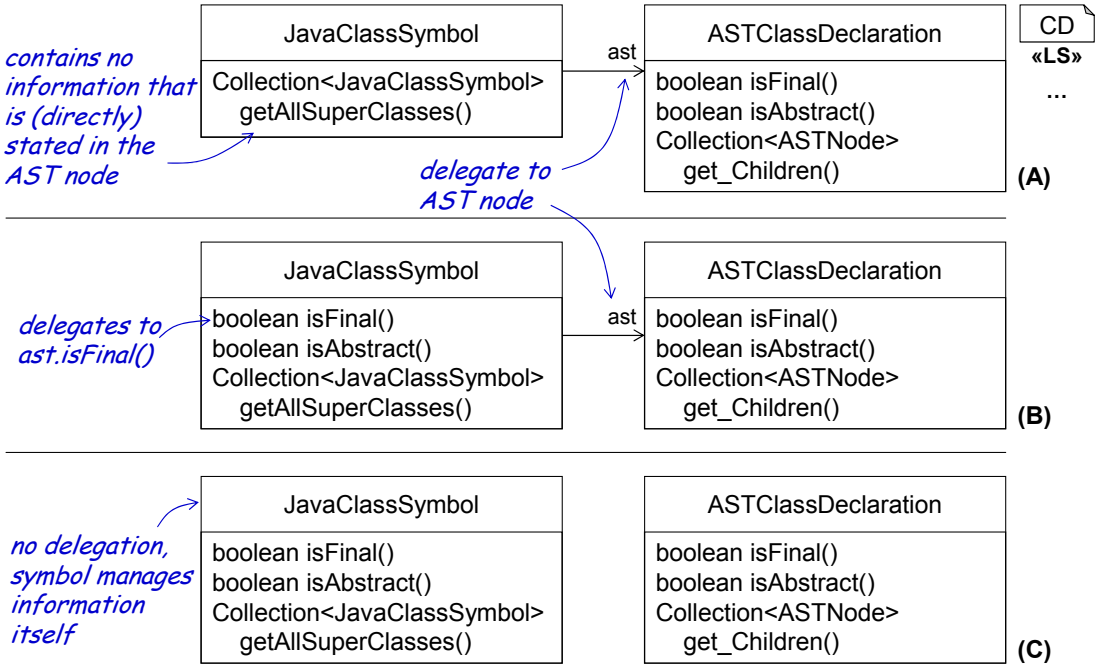


Figure 4.8: General idea of the patterns *(A) Symbol Provides No Information Directly Contained in Related AST Node* (top part), *(B) Symbol Provides Information of Related AST Node Via Delegation* (middle part), and *(C) Symbol Provides Information of Its AST Node Without Delegation* (bottom part).

(A) Symbol Provides No Information Directly Contained in Related AST Node

As depicted in the top part of Figure 4.8, this pattern strictly separates information of a symbol and its related AST node. While the AST node contains information directly specified in the model, the symbol only provides further information that is not *directly* stated in the model. The method `getAllSuperClasses` of `JavaClassSymbol` (cf. top part Figure 4.8), for instance, returns all superclasses of a Java class by computing the transitive closure, *derived* from the specified information. To enable access to the AST node's information, the symbol class must provide a respective public method.

The benefit of this implementation pattern is that it is lightweight and easy to implement. Moreover, it ensures consistency between a symbol and its AST node by preventing redundancy between these. However, these benefits come with some major drawbacks. First, the AST does not always provide the information in a convenient way, resulting from the fact that its structure is determined by a grammar definition (cf. Section 2.2.2). Second, it strongly increases the adaption effort for language composition since then not only the symbol needs to be adapted but also the AST node (cf. Chapter 8). Third,

in order to reuse the symbol, e.g., via language inheritance (cf. Section 8.4), a subtype of the AST node is required as well. As a consequence, it enforces two languages with the same essential model to be in a grammar hierarchy (cf. automaton grammars in Section 7.3). Finally, this implementation pattern does not allow to build and use the symbol table independently from the AST (cf. Section 4.6). This case will probably occur often when dealing with modeling in the large.

(B) Symbol Provides Information of Related AST Node Via Delegation

In this pattern the symbol internally delegates to its related AST node. In contrast to the previous pattern, the AST node is encapsulated in the symbol which itself provides the required information. The middle part of Figure 4.8 exemplifies this pattern. As it can be seen, `JavaClassSymbol` provides some information that is also part of the respective AST node `ASTClassDeclaration`, such as `isFinal` which delegates to `ast.isFinal()`.

Similar to the pattern *(A) Symbol Provides No Information Directly Contained in Related AST Node*, this pattern ensures consistency between symbol and AST node via delegation. Furthermore, it enables to abstract from information of the AST node and to provide only *essential* information required for the symbol table user in a convenient way. Moreover, encapsulating the AST node simplifies the adaption effort when composing languages (cf. Chapter 8) since only the symbol has to be adapted. Same as the previous case, this pattern requires a subtype of the AST node, and thus, hinders using the symbol independently from its AST node.

(C) Symbol Provides Information of Its AST Node Without Delegation

The last pattern (bottom part of Figure 4.8) omits the AST node completely[4] and provides each *essential* information (that is part of the language's essence) explicitly. That way, similar to the previous pattern, it hides information of the AST that is not relevant for the symbol table user. Also, the adaption effort is reduced. Its main advantage over the two previous patterns is that it strictly separates symbols and AST nodes and enables using each without the other. This allows to reuse the symbol in other languages without enforcing a subtype of its AST node. Although the symbol is created mostly based on the AST (cf. Chapter 5), it does not *statically* depend on the concrete type of the AST node, such as `ASTClassDeclaration`. The drawback of this pattern is that the redundant information introduced in the symbol increases both development effort and maintenance effort.

[4]Please note that this does not include the generic relation between AST nodes and symbols, presented in Section 4.6.

Discussion

While the first two presented patterns ensure consistency between a symbol and its related AST node *by construction*, the last pattern simplifies reuse and extension because of (type) independence between these structures. As a consequence, the last pattern *(C) Symbol Provides Information of Its AST Node Without Delegation* facilitates (a-posteriori) language composition (cf. [HLMSN+15a]).

Since the first two patterns are easier and faster to realize, they can be used for rapid prototyping in an agile process. Subsequently, the third pattern can be applied iteratively and incrementally. Furthermore, combinations of the patterns are possible. For example, when combining pattern *(B) Symbol Provides Information of Related AST Node Via Delegation* (with optional AST node) and pattern *(C) Symbol Provides Information of Its AST Node Without Delegation* allows to retrieve information from the AST node if it is not locally stored in the symbol table.

4.1.2 Patterns for Symbols Representing Similar Model Elements

Some software languages provide model elements that are similar in many ways. In Java, for example, classes, interfaces, and enums have a lot in common, e.g., having supertypes, fields, and modifiers. However, they still yield some differences. Interfaces, for example, cannot extend a class. Moreover, enums and classes may *implement* an interface, while an interface may *extend* other interfaces.

In those cases, it is not always clear how the corresponding symbol table structure should look like, for instance, whether each kind of Java type should be represented by its own symbol or not. For this reason, the current section presents two possible approaches by the example of Java types and discusses their advantages as well as disadvantages.

(D) Same Symbol Class for Similar Model Elements

The usage of classes, interfaces, and enums in Java is syntactically the same. For example, when declaring a Java field `T f`, it is not necessary to know whether `T` is a class, an interface, or an enum. Hence, when searching for `T`, it should be sufficient to state that it is a *Java type* named `T`. The differences between these types are based on the context. For example, interfaces cannot be instantiated, hence, statements like `I i = new I()` are not allowed for them. These cases can be checked by context conditions together with a type system (cf. Section 2.2). A single symbol class can group the three Java types as, for example, `JavaTypeSymbol` depicted in the left part of Figure 4.9. In order to distinguish between these types, `JavaTypeSymbol` must provide respective methods, such as `isInterface` and `isEnum`.

The main advantage of this pattern is that the usage of the different types is unified, and thus, does not need to be distinguished explicitly. This is especially helpful during the symbol table creation phase where concrete information about a referenced symbol

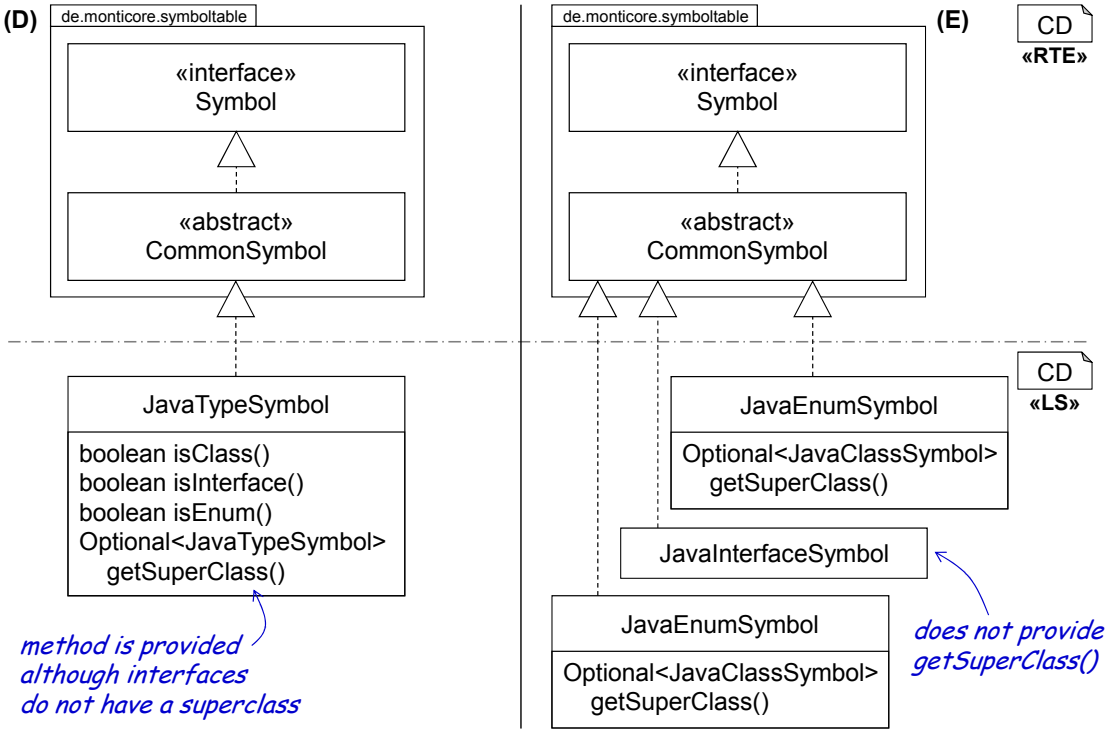


Figure 4.9: General idea of the patterns *(D) Same Symbol Class for Similar Model Elements* (left part) and *(E) Different Symbol Classes for Similar Model Elements* (right part).

is not (yet) available, as discussed in Chapter 5. Another advantage of this pattern is that the symbol structure is less complex (only one symbol class for three types). This, among others, reduces the effort for extending a language (cf. Section 8.4).

However, the main disadvantage is that the symbol class which represents similar model elements must provide an interface with a superset information of all types. For instance, a class and an enum have a superclass, but an interface does not. Nevertheless, `JavaTypeSymbol` must provide the method `getSuperClass` and, for example, throw an exception or return an empty value if the represented type is an interface.

(E) Different Symbol Classes for Similar Model Elements

A further possible symbol structure for Java types is employing dedicated symbol classes for each type, that means, class, interface, and enum. The right part of Figure 4.9 highlights this case. The classes `JavaClassSymbol`, `JavaInterfaceSymbol`, and `JavaEnumSymbol` represent the corresponding Java types.

The main advantage and disadvantage are conversed to the previous pattern *(D)*

Same Symbol Class for Similar Model Elements (cf. Section 4.1.2). That means, this pattern enables to specify a symbol class that perfectly fits the model element it denotes. For example, `JavaInterfaceSymbol` omits the `getSuperClass` method mentioned in the previous pattern. However, the usage of the different types is not unified, and hence, needs to be known during the symbol table creation phase. Furthermore, the symbol structure becomes more complex. In the Java example shown on the right part of Figure 4.9, three symbol classes are required, instead of one.

Discussion

Which pattern is appropriate, highly depends on the semantics of the language as well as how the language engineer intends the symbols to be used by, e.g., generator developers. The following questions can help to choose the more appropriate pattern:

- **Are the different model elements conceptually similar, i.e., do they provide similar information?** A language specification (if exists) can help to answer this question. The Java language specification [GJS+14], for example, states that a class declaration is either a normal class declaration or an enum declaration. Also, the grammar rules are similar; both have a `ClassBody`. Hence, classes and enums are conceptually equal.
- **Is it possible to (syntactically) distinguish between the usages of the different model elements?** If not, the model elements probably have a lot in common. As already mentioned, the field declaration `T f` does not state whether `T` is a class, an interface, or an enum.

If the first question can be answered with "yes" and the second with "no", a unified symbol class is useful, i.e., pattern *(D) Same Symbol Class for Similar Model Elements* (cf. Section 4.1.2). In contrast, if the answers are "no" and "yes", respectively, dedicated symbol classes are better suited, i.e., pattern *(E) Different Symbol Classes for Similar Model Elements* (cf. Section 4.1.2). Else, a combination of the presented patterns might sometimes be appropriate. For example, following from the Java language specification [GJS+14], methods and fields both are class members having some aspects in common, e.g., both have modifiers and are defined in a class. However, their usage is distinguished syntactically, e.g., `C.f` or `C.m()`. Hence, both questions above are answered with "yes". Therefore, it can be useful to introduce a (abstract) class `JavaClassMemberSymbol` which then is subclassed by the concrete symbols `JavaFieldSymbol` and `JavaMethodSymbol`.

Whether two model elements are similar, however, depends on the language engineer. In the Java class member example the language engineer might be of the opinion that fields and methods are not similar at all, and hence, answers the first question with "no". As a consequence, pattern *(E) Different Symbol Classes for Similar Model Elements* (cf. Section 4.1.2) would apply.

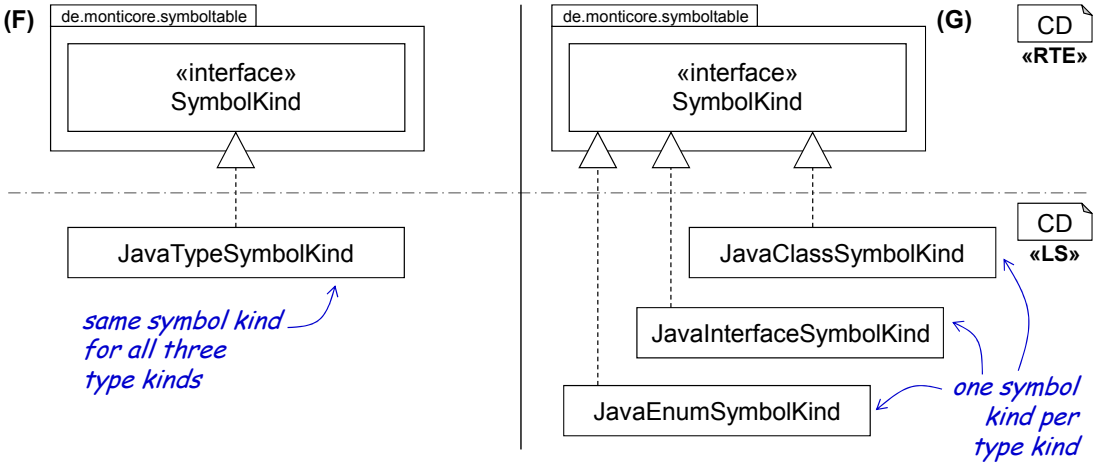


Figure 4.10: General idea of the patterns *(F) Same Symbol Kind for Similar Model Elements* (left part) and *(G) Different Symbol Kinds for Similar Model Elements* (right part).

4.1.3 Patterns for Symbol Kinds of Similar Model Elements

Same as for symbols, different patterns exist for symbol kinds of similar model elements (cf. Figure 4.10). The advantages and disadvantages are similar to the ones introduced in Section 4.1.2. Hence, this section briefly introduces the patterns for symbol kinds of similar model elements.

(F) Same Symbol Kind for Similar Model Elements

In particular, symbol kinds are employed in the resolution process (cf. Chapter 6) and for the translation process when composing models (cf. Chapter 8). Consequently, using one mutual symbol kind for different model elements (left part of Figure 4.10), allows to easily search for all matching symbols representing the different model elements.

(G) Different Symbol Kinds for Similar Model Elements

In contrast to the previous pattern, separate symbol kinds (right part of Figure 4.10) enable a more specific resolution and translation process. At the same time, they increase the complexity of the symbol kind structure and may increase the development effort.

Discussion

In general, (at least) one specific symbol kind should exist for each specific kind of symbols, and thus, the chosen pattern for symbols determines the appropriate pattern for

symbol kinds. If, for example, the `JavaTypeSymbol` class existed (pattern *(D) Same Symbol Class for Similar Model Elements*, cf. Section 4.1.2), the corresponding symbol kind `JavaTypeSymbolKind` should be created (pattern *(F) Same Symbol Kind for Similar Model Elements*). Else, if a dedicated symbol class for each Java type existed (pattern *(E) Different Symbol Classes for Similar Model Elements*, cf. Section 4.1.2), dedicated classes for the corresponding symbol kinds should also exist (pattern *(G) Different Symbol Kinds for Similar Model Elements*).

4.1.4 Patterns for Relating a Symbol and Its Kind

After introducing patterns for symbols and symbol kinds separately in Sections 4.1.2 and 4.1.3, this section discusses three patterns for (technically) relating a language-specific symbol and its kind. The patterns are introduced by the example of Java field symbols.

(H) Separating a Symbol and Its Kind into Different Classes

In Figure 4.11 dedicated classes exist for the Java field symbol and its kind. The interface of the symbol (i.e., its methods) is stated in the `JavaFieldSymbol` class. `JavaFieldKind` implements `SymbolKind`'s methods similar to Listing 4.4.

The advantage of this approach is that the symbol and its kind are not mixed up which simplifies reuse and adaptions, especially when composing languages (cf. Chapter 8). Furthermore, the symbol kind class can be fully generated as described in Section 7.5 since it only provides implementations for methods of `SymbolKind`.

The main disadvantage of this pattern is that symbols with the same kind are not enforced to have the same (subset of) methods. Figure 4.12 shows an example. Both languages Java and C# have fields with similar concepts such as public and private mod-

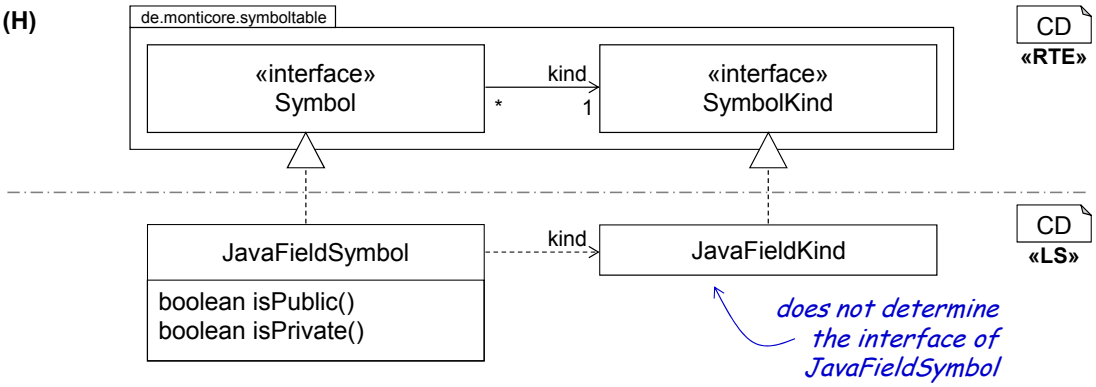


Figure 4.11: General idea of the pattern *(H) Separating a Symbol and Its Kind into Different Classes.*

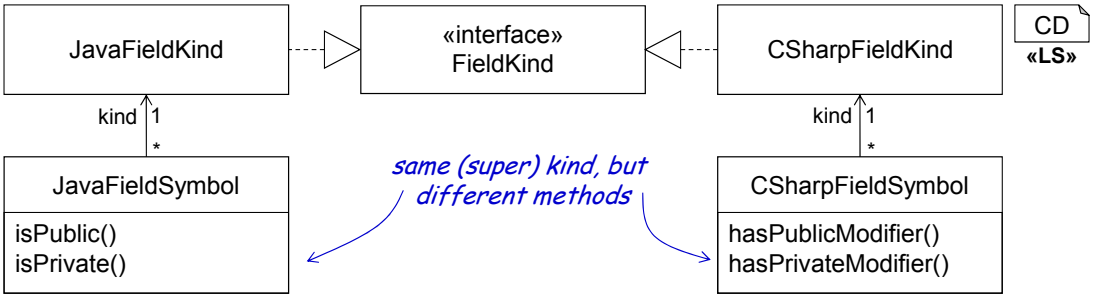


Figure 4.12: Applying the pattern *(H) Separating a Symbol and Its Kind into Different Classes* to a symbol kind hierarchy.

ifiers. Hence, `JavaFieldKind` as well as `CSharpFieldKind` implement `FieldKind`. But, as it can be seen, the respective symbols provide different methods for the same information, e.g., `isPublic` and `hasPublicModifier`.

(I) Symbol Class Implements Kind Interface

To ensure that symbols of the same kind provide the same interface, the pattern shown in Figure 4.13 is suitable. Here, `JavaFieldKind` is an interface instead of a class (cf. Figure 4.11). Furthermore, it specifies the two methods `isPublic` and `isPrivate`. Symbols of that kind, such as `JavaFieldSymbol` must implement `JavaFieldKind`. That way, it is statically ensured that all implementing symbols provide the same methods.

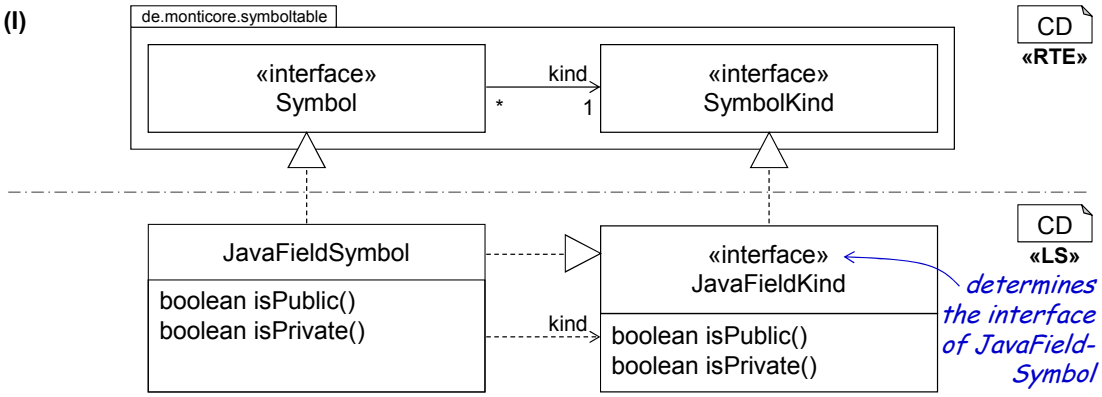


Figure 4.13: Example of pattern *(I) Symbol Class Implements Kind Interface*. The symbol kind is implemented as an interface and specifies methods each symbol of that kind must provide.

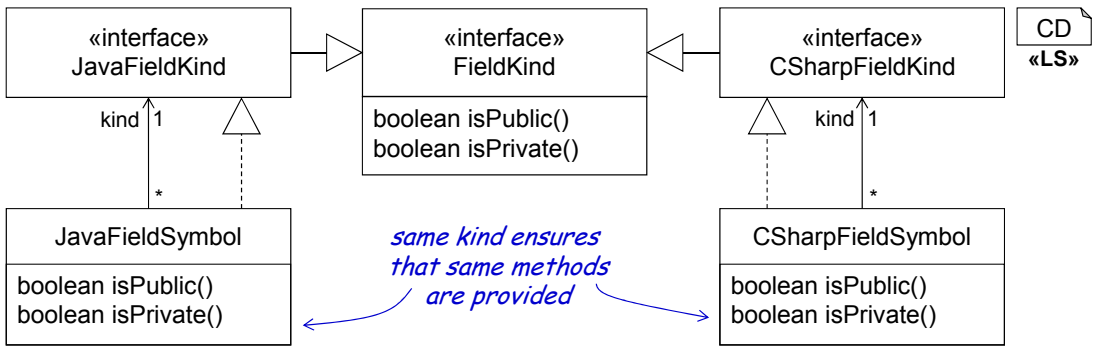


Figure 4.14: Applying the pattern *(I) Symbol Class Implements Kind Interface* to a symbol kind hierarchy.

Figure 4.14 illustrates how this pattern avoids the drawback of pattern *(H) Separating a Symbol and Its Kind into Different Classes*. Since `FieldKind` specifies the methods of field kinds, both symbols `JavaFieldSymbol` and `CSharpFieldSymbol` provide exactly the same method signatures. Hence, whenever a `FieldKind` is requested, any of these symbols can be used.

This pattern yields some disadvantages itself. First, in contrast to *(H) Separating a Symbol and Its Kind into Different Classes*, a symbol kind in this approach cannot be fully generated (cf. Section 7.5) since it is more complex and its methods are language-specific. Second, determining the interface of all symbols of a specific kind might be to restrictive and may hamper the independent engineering of languages.

(J) Same Class For a Symbol and Its Kind

Figure 4.15 demonstrates a third pattern for implementing a symbol and its kind. In contrast to the previous patterns, symbol and kind are not implemented by separated classes (or interfaces). Hence, `JavaFieldSymbol` represents both, a Java field symbol and its kind. The main advantage of this approach is that it is simple and ensures that all symbols in a hierarchy have the same kind and provide the same information (same as the pattern *(I) Symbol Class Implements Kind Interface*). However, it lacks separation of concerns which impedes both reuse and the generation process (cf. Chapter 7).

Discussion

In a sum, the first two patterns have a bias towards the last pattern. They simplify reuse as well as a (partial) generation of the symbol kind (cf. Section 7.5). Furthermore, they facilitate language composition. Although the last pattern is simple and reduces complexity (only one class for symbol and its kind), it should only be applied if a later composition with other languages can be excluded.

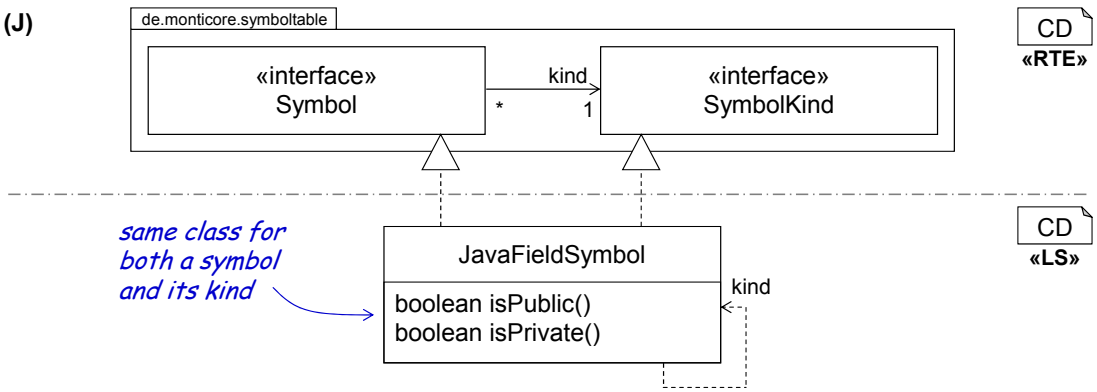


Figure 4.15: General idea of the pattern *(J) Same Class For a Symbol and Its Kind.*

4.2 Technical Realization of Scopes

Figure 4.16 shows an overview of the technical scope classes for the scope types introduced in Section 3.5. `Scope` represents the superinterface of all scopes and, among others, enables the enclosing-sub relation leading to a graph of scopes (cf. Section 3.5).

While dedicated classes for artifact scopes (cf. Def. 3.10) and the global scope (cf. Def. 3.11) exist, named and unnamed scopes (cf. Def. 3.9) as well as shadowing and visibility scopes (cf. Def. 3.8) are realized via `Scope`'s methods `getName` and `isShadowingScope`, respectively:

getName() In case the scope is named, this method returns its corresponding name, else, i.e., if the scope is unnamed the value is absent. A scope that is spanned by a symbol has the same name as that symbol. This default behavior only applies if the symbol's name is not explicitly set via `setName` (cf. Section 4.2.1).

isShadowingScope() Depending on whether the scope is a shadowing scope or a visibility scope, this method returns `true` or `false`. By default, the scope is a shadowing scope if it has a name, i.e., `getName` returns a non-empty value which in particular is the case for scopes spanned by symbols.

Furthermore, `Scope` provides the method `exportsSymbols` which states whether a scope exports (or forwards) its symbols so that other scopes can explicitly import them (cf. Section 3.10). Named scopes standardly export their symbols. Chapter 6 discusses the role of this method during the resolution process.

`CommonScope` provides implementations for the three methods described above. In contrast to `CommonSymbol` (cf. Figure 4.2), `CommonScope` is not an abstract class. This is because a scope does not necessarily require language-specific information while a symbol relies on language-specific information.

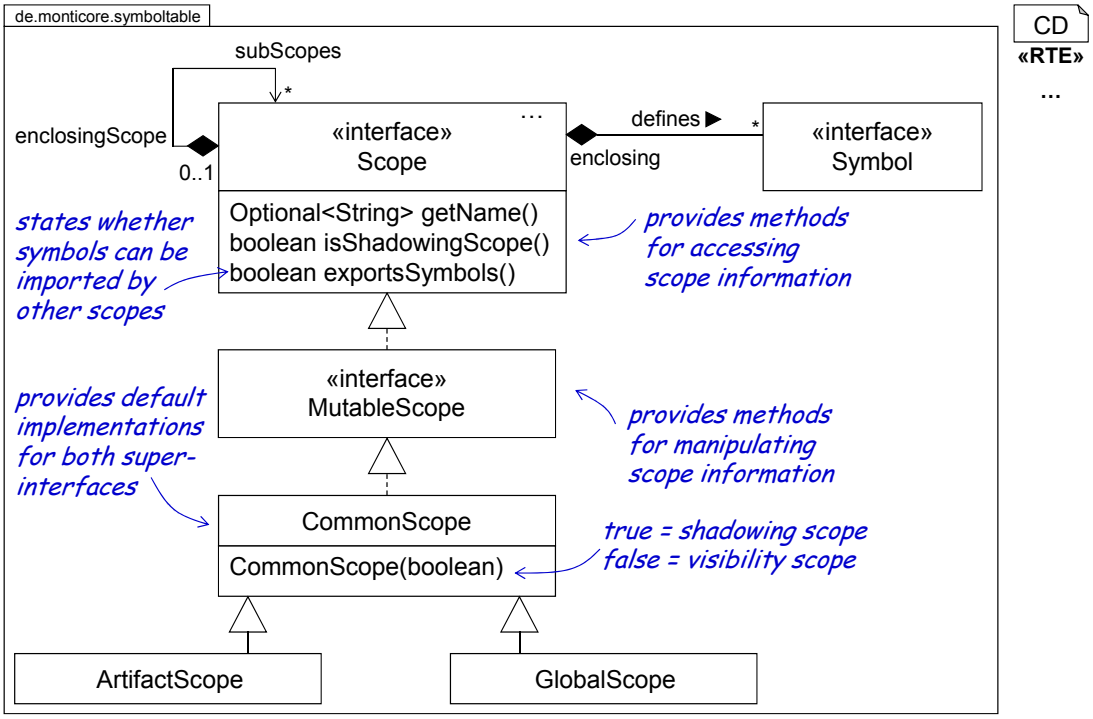


Figure 4.16: Overview of the technical classes for the scope types introduced in Section 3.5, i.e., named and unnamed scopes, visibility and shadowing scopes, artifact scopes, and the global scope.

The classes `ArtifactScope` and `GlobalScope` represent an artifact scope and the global scope, respectively. Both classes subclass `CommonScope` and override some methods in order to adjust the resolution process. These are discussed together with the resolution process in Chapter 6.

Similar to our approach, Parr [Par10] provides a `Scope` interface and its default implementation `BaseScope` to access, among others, a scope's name and its enclosing scope. Also, a scope starts the symbol resolution. The `GlobalScope` in [Par10] does not introduce additional functionality. It mainly differs from other scopes by its name (i.e., "global"). In contrast, the `GlobalScope` class of the SMI is essential for inter-model resolution and model loading (cf. Section 6.7 and Section 6.9). Namespaces in [Völ11] are language-unspecific and unnamed symbol table containers, represented by the `NameSpace` class. There are no subclasses (in the provided infrastructure), e.g., for the global scope. In Xtext [Bet13] a scope determines which elements of a model (element) are visible for a specific reference (cf. Section 3.5). A scope provider conducts this task and returns a list of scopes which, among others, specifies the order to search for the visible elements. Same as Xtext, a scope in MPS [VS10] focuses on a specific point in the model and is determined by so-called scope providers.

4.2.1 MutableScope: Interface for Manipulating a Scope

The `Scope` interface solely provides methods for *accessing* a scope's information, e.g., its containing symbols. For symbol table users, such as generator engineers and context condition engineers, this is sufficient since they do not manipulate the scopes but only access the contained symbols. However, during the symbol table creation phase (cf. Chapter 5) the language engineer needs to build up the scope graph and fill it with symbols. To enable the manipulation of scopes, SMI provides the interface `MutableScope`, as depicted in Figure 4.17. This interface provides operations that language engineers require but usually not language users. Since all scopes first have to be created as well as initialized before their usage (cf. Chapter 5), all scopes must be subtypes of `MutableScope`. Consequently, `CommonScope`—which serves as default implementation for scopes—subtypes `MutableScope` (cf. Figure 4.16).

Using a dedicated interface for manipulating the scope has the advantage that the `Scope` interface is not polluted with information that is irrelevant most of the time (cf. *Interface Segregation Principle* [Mar02]). Furthermore, separating the read and write methods improves the tool support for symbol table users. For example, the auto-completion functionality provided by IDEs such as JetBrains' IntelliJ IDEA [www16g] and Eclipse [www16c] will only list methods relevant for accessing information from the scope. In order to access a scope as an instance of `MutableScope`, the `Scope` interface provides the method `getAsMutableScope`. This not only obviates the need for type castings from `Scope` to `MutableScope`, but also ensures that each scope is a subtype of `MutableScope`.

Figure 4.17 depicts the methods of `MutableScope` that enable manipulating a scope. In addition to these methods, `MutableScope` provides methods that are important for the (internal) resolution process. Chapter 6 elaborates those methods in detail. The following describes the default implementations for the presented methods as provided by `CommonScope`:

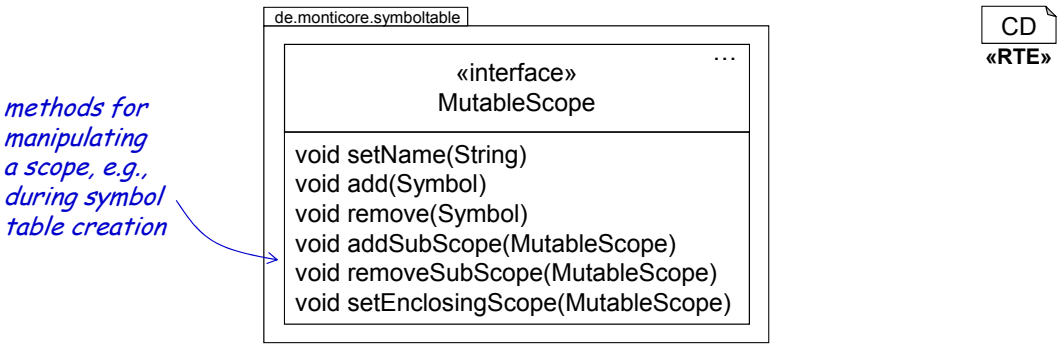


Figure 4.17: Methods provided by the `MutableScope` interface for manipulating a scope.

setName(String) Sets the name of the scope, and thus, disables the default calculation as described earlier in this section.

add(Symbol) Adds a symbol to the scope. Also, the scope is set as the symbol's enclosing scope.

remove(Symbol) Removes the given symbol from the scope and also unsets its enclosing scope relation.

addSubScope(MutableScope) This method adds a new subscope s to a scope e, if e does not already contain s. Also, e is set as enclosing scope of s using `setEnclosingScope(MutableScope)`. That way, the enclosing-sub relation remains consistent.

removeSubScope(MutableScope) This method removes a subscope s from a scope e and unsets the enclosing scope of s.

setEnclosingScope(MutableScope) Sets the enclosing scope e of a scope s. Also, s is added to e as a subscope (if not already contained).

Please note that the last three methods for setting the enclosing-sub relation of scopes (cf. Section 3.5) require a `MutableScope`, not solely a `Scope`. This ensures that the described manipulations can be conducted. This, as mentioned above, requires that all scopes are not only a subtype of `Scope` but also a subtype of `MutableScope`. Following from this, each method returning a scope—e.g., the `getEnclosingScope` method of the `Scope` interface—returns a subtype of `MutableScope`.

While it is useful to employ separate interfaces for accessing and modifying a scope, it would not be reasonable for symbols. The reason is that scopes can be used completely generically, like `scope.resolve(...)`, and hence, are declared as `Scope s;` instead of, for example, `JavaTypeScope s;`. That way, it is ensured that methods defined in `MutableScope` (and its subtypes) are hidden from the language user. In contrast, a symbol is strongly language-specific. Thus, a generic declaration like `Symbol s;` does not allow to retrieve the required information. Instead a language-specific declaration as `JavaTypeSymbol s;` has to be specified.

4.2.2 Scopes as Repositories for Symbols

A repository [Fow03, Eva03] stores elements and provides ways for retrieving them. Furthermore, elements can be added or removed (if not read-only). An important task of a repository is to encapsulate how elements are managed and persisted. The client solely has to request the needed information.

Consequently, a scope in SMI can be considered as a *repository for symbols*. It allows to add and remove symbols (via `MutableScope`) and also to retrieve them (via `Scope`).

If a requested symbol is not found in the current scope, the search can be continued in the enclosing scope until the symbol is found or the root scope is reached [ALSU06] (cf. Chapter 6). Furthermore, models contained in artifacts are loaded (from the file system), if necessary. The whole process is completely encapsulated within the scope graph and hidden from the symbol table user. In this section, the role of a single scope is described which serves as a basis for the collaboration of scopes elucidated in Chapter 6.

Figure 4.18 shows the conceptual structure of a (Java or C#) class scope, which is a repository for methods and fields. It contains two method symbols (m_1 and m_2) and two field symbols (f_1 and f_2). A user can request symbols in different ways, for example, by stating the symbol kind (*method*) and the name (m_1). Also, all symbols of a specific kind (e.g., all methods) can be obtained.

As depicted in Figure 4.19, the `Scope` interface provides three methods to obtain the symbols it contains *locally*, i.e., not imported from other scopes (cf. Section 3.10). The first method `getLocalSymbols` returns a map of all symbols that are defined directly in the scope. The map groups same-named symbols, i.e., $name \rightarrow \{symbols\}$. In the example highlighted in Figure 4.18, `getLocalSymbols` returns all fields and methods. The second method, `resolveLocally(String, SymbolKind)`, resolves a specific symbol by its name and kind.

Analogously, `resolveLocally(SymbolKind)` resolves all symbols with the specified symbol kind. In contrast to `getLocalSymbols`, the two `resolveLocally` methods make use of so-called resolving filters. Given a collection of symbols, a *resolving filter* filters all symbols whose name and kind do match the requested ones.

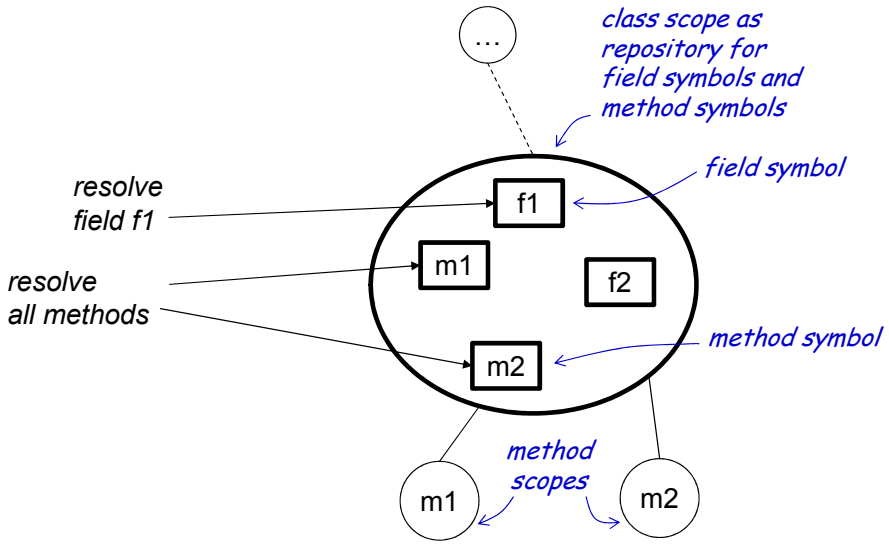


Figure 4.18: The conceptual structure of a scope which serves as a repository for symbols.

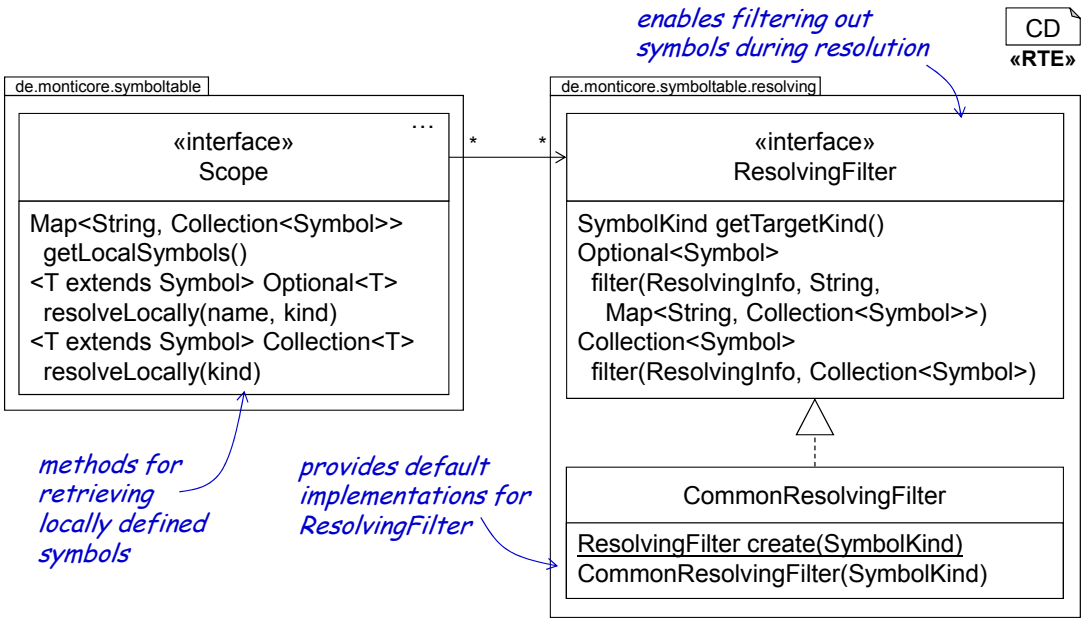


Figure 4.19: Excerpt from the methods the `Scope` interface provides for retrieving its locally defined symbols. For its `resolveLocally` methods `Scope` makes use of resolving filters represented by the same-named interface.

As shown in Figure 4.19, resolving filters are realized by a same-named interface which defines three methods. Their default behavior provided by `CommonResolvingFilter` is as follows:

getTargetKind() Specifies the symbol kind the resolving filter searches for. Symbols with different kinds are filtered out.

filter(ResolvingInfo, String, Map<String, Collection<Symbol>>) Searches in the given map of symbols (third parameter) for a symbol with the name specified in the second parameter and the symbol kind, as specified in `getTargetKind`. The symbols locally defined in a scope—retrieved via `getLocalSymbols` (see above)—are typically passed as third parameter. The first parameter provides additional information which is important essentially for adapted resolving (cf. Section 8.2.2). Symbol kinds are compared via `SymbolKind`'s `isKindOf` method (cf. Section 4.1). If more than one matching symbol is found, a `Resolved-SeveralEntriesException` exception is thrown.

filter(ResolvingInfo, Collection<Symbol>) Given a collection of symbols, this method returns all symbols whose symbol kind match the kind specified in `getTargetKind`. Since this method filters the symbols only based on their kind, no map is required as in the previous method. Again, the first parameter is utilized for adapted resolution.

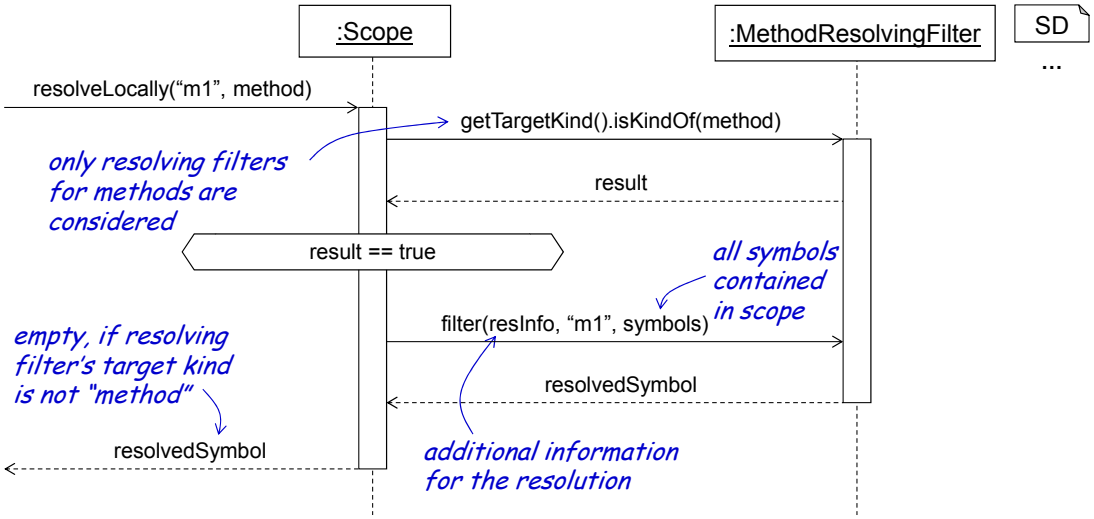


Figure 4.20: The default procedure for filtering out symbols via resolving filters.

Please note that while `filter(ResolvingInfo, Collection<Symbol>)` returns a collection of symbols, the other `filter` method returns (at most) one symbol. The reason is that it is quite common that symbols of the same kind (e.g., fields) are defined in the same scope. In contrast, not many languages allow multiple definitions of same-named symbols with the same kind within the same scope. Hence, the first `filter` method treats those cases as an exception.

`CommonResolvingFilter` provides the method `create(SymbolKind)` in order to easily instantiate a resolving filter for a specific symbol kind. That way, `CommonResolvingFilter` does not have to be subclassed if only the default behavior is required. Figure 4.20 illustrates the default filtering process by an example where a method symbol named `"m1"` is resolved locally.

The scope first checks whether the resolving filter's target kind matches the requested kind, that is, `method`. If this is the case, the method name `"m1"` together with all `symbols` contained in the scope are passed to the `filter` method of the resolving filter which then filters out all unmatched symbols, i.e., symbols that either are not kind of `method` or are not named `"m1"`. Put another way, the resolving filter searches for all method symbols named `"m1"`.

Resolving filters are essential in the SMI. First, they enable kind-based resolution including kind hierarchies (cf. Chapter 6). That way, among others, same-named symbols with different kinds can coexist. Depending on the specified kind, the other symbols are filtered out. Second, resolving filters allow to customize each scope individually, i.e., if no resolving filter for a specific kind exists, that scope cannot resolve symbols of that kind. Furthermore, resolving filters enable symbol adaption when composing languages. Their role in language composition is elucidated in Chapter 8.

Similar to our approach, the `Scope` interface in [Par10] provides a `resolve` method for resolving a symbol. However, its only parameter is the symbol's name since symbol kinds do not exist (explicitly). Consequently, resolving filters are not required. Völkel [Völ11] provides so-called *resolvers* for conducting kind-based resolutions. In contrast to resolving filters as introduced in the current thesis, resolvers are responsible for both traversing the scope hierarchy and filtering out the symbols. This, however, does not ensure that all symbols in a scope behave the same regarding their resolution [NTVW15] (cf. Section 3.5), for example, if different resolvers traverse the scope graph with different strategies.

4.3 Technical Realization of Scope Spanning Symbols

The `ScopeSpanningSymbol` interface depicted in Figure 4.21 represents scope spanning symbols, as defined in Def. 3.12. It extends `Symbol` and additionally spans a `Scope`. While a scope spanning symbol always spans a scope (cardinality `1`), a scope is not necessarily spanned by a symbol (cardinality `0..1`). The scope of an if block, for example, is not spanned by any symbol. The bottom part of Figure 4.21 highlights that the `JavaTypeSymbol` (e.g., depicted in Figure 4.9 on page 62) is not only a subtype of `Symbol` but also a subtype of `ScopeSpanningSymbol`. Furthermore, it spans a `JavaTypeScope`.

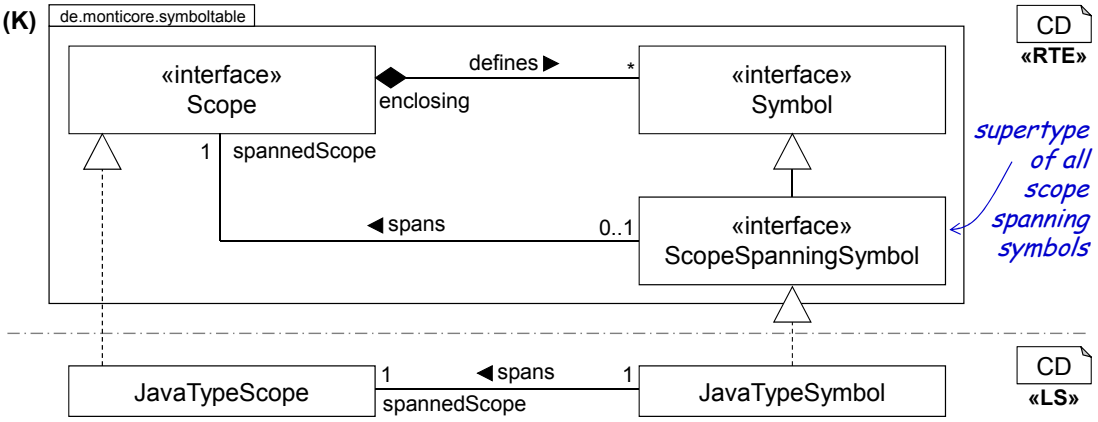


Figure 4.21: Overview of the `ScopeSpanningSymbol` interface.

Same as for `Symbol`, a common class exists for `ScopeSpanningSymbol`, namely `CommonScopeSpanningSymbol` (cf. Figure 4.22), which extends `CommonSymbol` and additionally provides the factory method [GHJV95] `createSpannedScope`. This method instantiates the spanned scope of the symbol which by default is a named

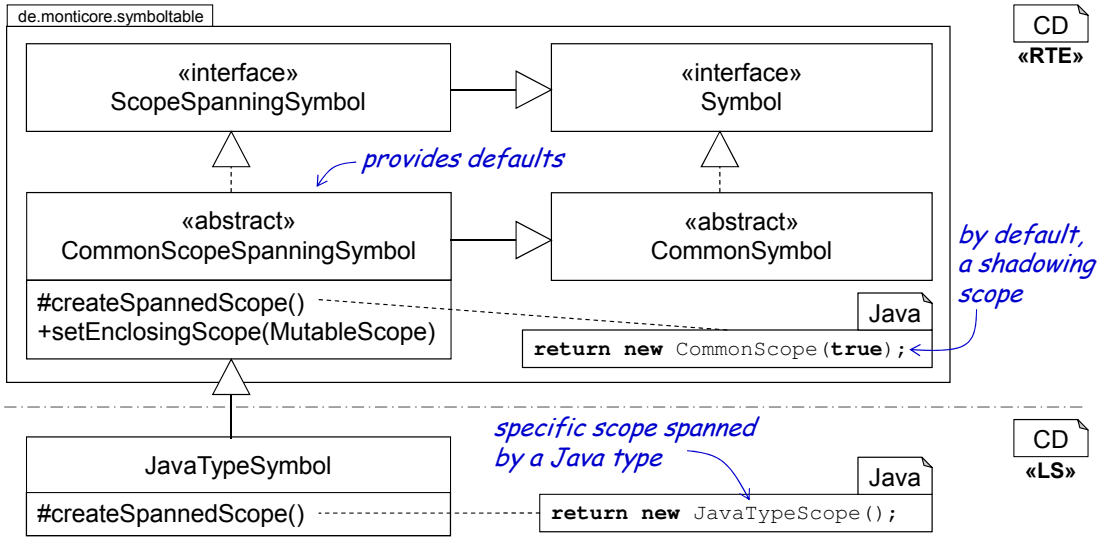


Figure 4.22: Implementation of a language-specific scope spanning symbol by the example of `JavaTypeSymbol`.

shadowing scope (cf. Section 4.2). In many cases, the generic `CommonScope` is sufficient since scopes typically behave the same, e.g., manage symbols and resolve names. However, some scopes require language-specific information. A class scope in Java, for example, imports the symbols of its enclosing artifact scope (language-unspecific) as well as (visible) exported symbols (cf. Section 3.10) of its superclass and implemented interfaces (language-specific). Hence, `JavaTypeSymbol` overrides `createSpannedScope` in order to create an `JavaTypeScope` instance[5] (cf. bottom part of Figure 4.22). Furthermore, `setEnclosingScope(MutableScope)` sets the enclosing scope of both the symbol and its spanned scope.

While a scope can be considered as a generic repository for symbols (cf. Section 4.2.2), its spanning symbol is rather a *language-specific view* on that scope and enables conducting more complex requests. It explicitly embodies the model's interface. Figure 4.23 shows the conceptual presentation of a symbol and its spanned scope, based on Figure 4.18. The symbol table user can focus on the spanning symbol, which provides language-specific information, such as all method symbols defined in the (spanned) class scope. For this, the spanning symbol delegates to its spanned scope via generic resolving requests like "get all symbols of kind *method*".

Symbols are defined and stored only in their enclosing scope, not in the spanning symbol itself. Instead, the spanning symbol delegates to its spanned scope. In contrast, Völkel [Völ11] stores the symbol table entries ("symbols" in our terminology) redundantly in both the namespaces and the (spanning) symbol table entries. This, above all, follows

[5] Alternatively, the Factory pattern [GHJV95] can be exploited for creating the spanned scope.

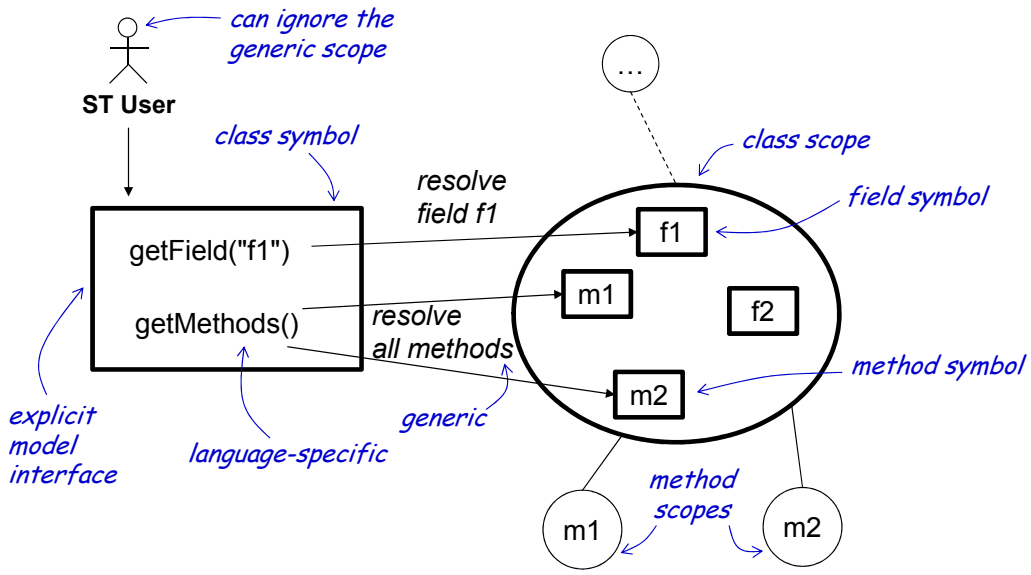


Figure 4.23: Conceptual structure of a generic scope (cf. Figure 4.18) spanned by a language-specific symbol. The symbol table user focuses on the language-specific symbol which delegates to its scope.

from the fact that no explicit concept for namespace spanning symbols exists, hence, their is no (direct) link between a symbol table entry and its spanned namespace (cf. Section 3.6). However, this can lead to inconsistencies between them, especially when composing languages, as discussed in Chapter 8. Consequently, a symbol table user must be aware of which source (i.e., symbol table entry or its spanned namespace) to use.

To ensure consistency, a scope spanning symbol in the current thesis delegates all requests to its spanned scope. Listing 4.24 exemplifies this by two methods of `JavaTypeSymbol`. The method `getField` (lines 4–7) returns a field symbol by its name. For this, `JavaTypeSymbol` solely delegates to its spanned scope using the generic method `resolveLocally(fieldName, JavaFieldSymbol.KIND)` (line 6). Since the generic aspects are encapsulated, the symbol table user can ignore them and instead focus on the language-specific method `getField`. The method `getMethods` in Listing 4.24 returns all method symbols defined in the Java type symbol (lines 9–17). Again, `JavaTypeSymbol` delegates to its spanned scope (line 11). Since `JavaMethodSymbol` represents both Java methods and Java constructors (assuming pattern *(D) Same Symbol Class for Similar Model Elements*, cf. Section 4.1.2), all constructors are filtered out (lines 14–16).

Please note that while the symbol resolution is conducted generically (line 11, Listing 4.24), filtering out constructors is language-specific (lines 14–16). For the former the default implementations provided by `CommonScope` are completely reused (including composition aspects). The language engineer only has to implement the language-specific

```java
public class JavaTypeSymbol
      extends CommonScopeSpanningSymbol {

  public Optional<JavaFieldSymbol> getField(String fieldName) {
    return getSpannedScope()
      .resolveLocally(fieldName, JavaFieldSymbol.KIND);
  }

  public Collection<JavaMethodSymbol> getMethods() {
    Collection<JavaMethodSymbol> methodsAndConstructors =
      getSpannedScope().resolveLocally(JavaMethodSymbol.KIND);
    // filter out constructors since they have
    // the same kind as methods.
    return methodsAndConstructors.stream()
      .filter(method -> !method.isConstructor())
      .collect(Collectors.toList());
  }
}
```

Java «LS»

Listing 4.24: Implementation of language-specific methods which retrieve symbols defined in the spanned scope via its generic `resolveLocally` method.

aspects manually. Again, the generic parts are hidden from the language user.

Figure 4.25 highlights the differences in usage for starting resolving with the infrastructure presented in [Völ11] (top part) and as presented in the current thesis (bottom part). `meth` in the top part of Figure 4.25 (line 1) is a method entry and `methNsp` (line 2) its respective namespace. The resolution starts by invoking the `resolve` method of the generic `resolver` provided by the framework which then delegates to language-specific resolver clients beginning in `methNsp` [Völ11]. As it can be seen, `meth` itself does not participate in this process (i.e., it is not used in line 2). Consequently, a resolution cannot be conducted if only `meth` is available. Therefore, the framework must ensure that both `resolver` and `methNsp` are available. In contrast, since the SMI links a symbol and its spanned scope (via `ScopeSpanningSymbol`), it enables symbol resolution without requiring further information, as shown in the bottom part of Figure 4.25. Here, `meth` provides access to its spanned scope which itself starts the resolution via `resolve` (line 4). Alternatively `meth.getField("f")` conducts the resolution, avoiding generic aspects (e.g., `resolve`) to be explicitly used (line 5). This is only possible because `getField` provides a language-specific interface and delegates to `meth`'s spanned scope, same as in Listing 4.24.

Parr [Par10] introduces the abstract class `ScopedSymbol` which is both a symbol and a scope. That way, consistency between a symbol and its spanned scope is preserved by construction. However, one disadvantage is that `ScopedSymbol` and `BaseScope` (see above) have redundant implementations. In the current thesis, SMI explicitly separates

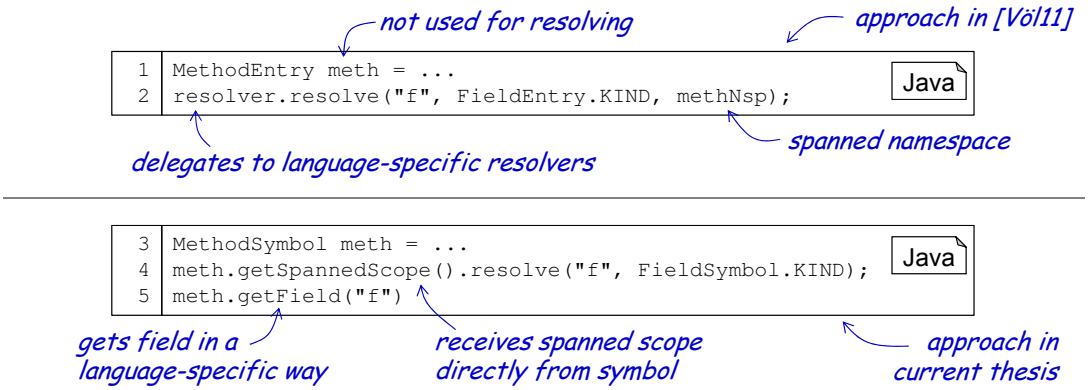


Figure 4.25: Usage for starting resolution process in [Völ11] (top part) and in the current thesis (bottom part).

symbols and scopes which simplifies reuse when composing languages and also facilitates customization of generated symbols and scopes (cf. Section 7.7). In contrast, Parr [Par10] focuses on implementing single languages (by hand). Xtext [Bet13] does not provide an explicit concept for elements spanning a scope. However, an `ENamedElement` (cf. Section 4.1) that contains other `ENamedElements` can be considered as such (even though it has no spanned scope).

4.3.1 Patterns for a Symbol and Its Spanned Scope

This section presents and discusses two patterns for implementing language-specific scope spanning symbols with SMI; either by using separated classes for the symbol and its spanned scope or by grouping them into the same class.

(K) Separating Symbol and Its Spanned Scope into Different Classes

Section 4.3 already introduced this pattern based on Figure 4.21 and Figure 4.22, i.e., `JavaTypeScope` and `JavaTypeSymbol` are both separated into their own classes. Although this pattern yields the drawback that it requires two classes, it should be applied for language-specific scope spanning symbols, for the following reasons:

- It explicitly separates the information provided by a symbol and the mechanism to resolve it (provided by the scope).
- It enables complete reuse of the default implementations provided by `CommonScope` and `CommonScopeSpanningSymbol` via class inheritance.
- It simplifies language inheritance since either class can be extended independently of the other.
- It facilitates the customization of generated symbols and scopes (cf. Section 7.7).

(L) Same Class for Symbol and Its Spanned Scope

SMI also allows for realizing a scope spanning symbol as suggested in [Par10] (cf. Figure 4.26). In this pattern, the same class represents both a symbol and a scope, i.e., it subtypes the `Scope` interface as well as the `ScopeSpanningSymbol` interface. In Figure 4.26 `JavaTypeSymbol` extends `CommonScopeSpanningSymbol` and implements `Scope`. Moreover, `JavaTypeSymbol` delegates to an instance of `CommonScope` to reuse the scope part. Alternatively, `JavaTypeSymbol` can extend `CommonScope`, implement `ScopeSpanningSymbol`, and delegate to `CommonScopeSpanningSymbol`. `JavaTypeSymbol`'s `createSpannedScope` method (and hence, also `getSpannedScope`) returns the same object but masked as a scope.

While this pattern is sufficient for a single language (as intended by Parr [Par10]), it hampers language inheritance and customization of generated symbols and scopes (cf. previous pattern).

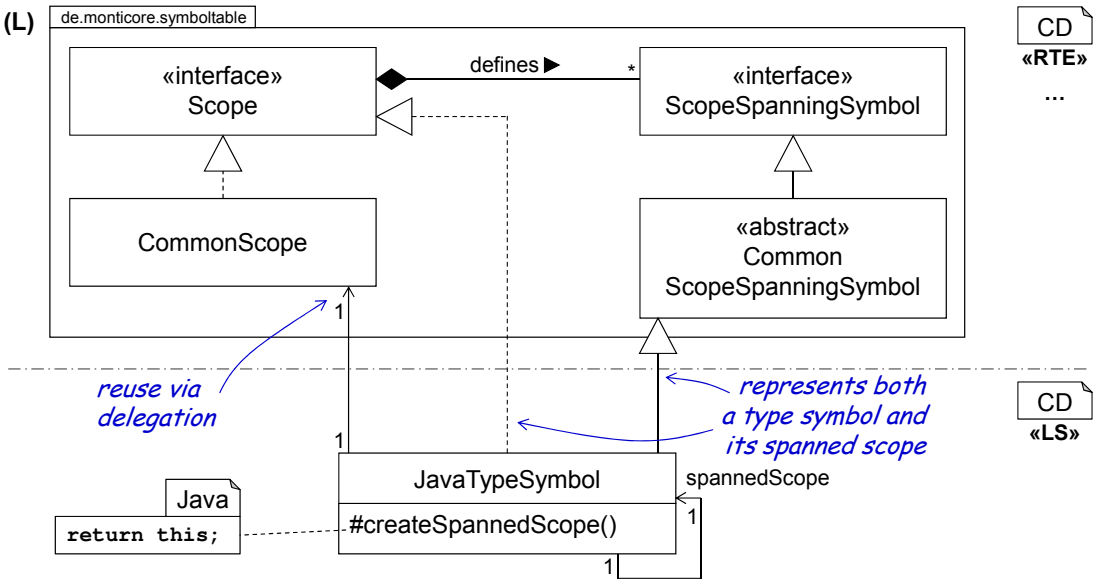


Figure 4.26: General idea of the pattern *(L) Same Class for Symbol and Its Spanned Scope*, following Parr [Par10].

4.3.2 Patterns for Symbols Representing a Parameterized Model Element that Spans a Scope

According to Aho et al. "[a]ll programming languages have a notion of a procedure" [ALSU06] which includes functions (as in C) and methods (as in Java and C#). In the current thesis we follow the term "methods" which "can behave like either functions or

procedures, but are associated with a particular class" [ALSU06]. Several possibilities exist for representing a method as a symbol and its spanned scope. This is because methods can define formal parameters as part of their signature as well as local variables within the method body, as exemplified in Listing 4.27.

```
1 public final class System {                                    Java
2   // ...
3   public static String getProperty(String key) {
4     checkKey(key);
5     SecurityManager sm = getSecurityManager();
6     // ...
7   }
8   // ...
9 }
```

Listing 4.27: Excerpt from method `getProperty` of class `java.lang.System`.

The method defines two variables, namely `key` (line 3) and `sm` (line 5). The former is a formal parameter whereas the latter is a local variable of the method. As it can be seen, the Java language (and many other languages, such as C, C#, and MontiArc [HRR12]) *syntactically* distinguishes between the definition of a formal parameter and the definition of a local variable, using round brackets (i.e., `(...)`) and curly brackets (i.e., `{...}`), respectively (cf. Section 3.5). However, this does not necessarily mean that a method spans two scopes and that the respective symbols are defined in different scopes. This observation based on the syntax not only applies to methods but to any (named) model element that spans a scope and further defines a parameter list, for example:

- A *generic Java class* defines type parameters which are, among others, visible within the class body [GJS+14].
- Similarly, *generic methods* define type parameters that can be accessed from within the method body.
- *Configurable component types* in MontiArc "define configuration parameters which represent variables with a certain data type. These parameters are used within the implementation of a component [...]" [Hab16].
- (Labeled) for-statements [GJS+14] (although usually not represented by a symbol) enable to define variables within an initialization section and refer to them from within the statements sections.

In order to simplify the wording, the following subsections introduce and discuss three patterns by the example of methods but can be applied for any of the above mentioned cases.

(M) Method Spanning a Parameter Scope and a Body Scope

Following the syntactical distinction of parameter definitions and local variable definitions, a method has two separated scopes, i.e., a parameter scope and a method body scope (cf. [Par10]). The classes `ParameterScope` and `MethodBodyScope` represent these scopes, as depicted in Figure 4.28.

Explicit distinction of the two scopes enables to easily retrieve parameters and local variables by resolving either in the parameter scope or in the method body scope. However, if it is not clear whether the requested symbol is a parameter or a local variable, the search must start in either one and continue in the other one. Hence, `ParameterScope` is the enclosing scope of `MethodBodyScope` (or vice versa). Furthermore, it must be determined whether `methodSymbol.getSpannedScope()` returns a `ParameterScope` instance or a `MethodBodyScope` instance. In the former case, i.e., `MethodSymbol` spans the `ParameterScope`, searching for a variable (either parameter or local variable) starts in the parameter scope and continues in the method body scope (i.e., top-down), even if the symbol is found (cf. Section 6.2). That way, ambiguous definitions are recognized. If the symbol is not found, the search continues in the class scope. In the latter case the search begins in the method body scope and continues in the parameter scope. Again, if the symbol is still not found, the search continues in the enclosing class scope. This case is more in line with the bottom-up resolution process which conducts the search from bottom to top (cf. Section 6.2).

However, the relation between parameter scope and class scope in neither approach is very intuitive. The subscope of a class scope is rather a method scope than a parameter scope since a class defines a method and not parameters. Furthermore, although separated,

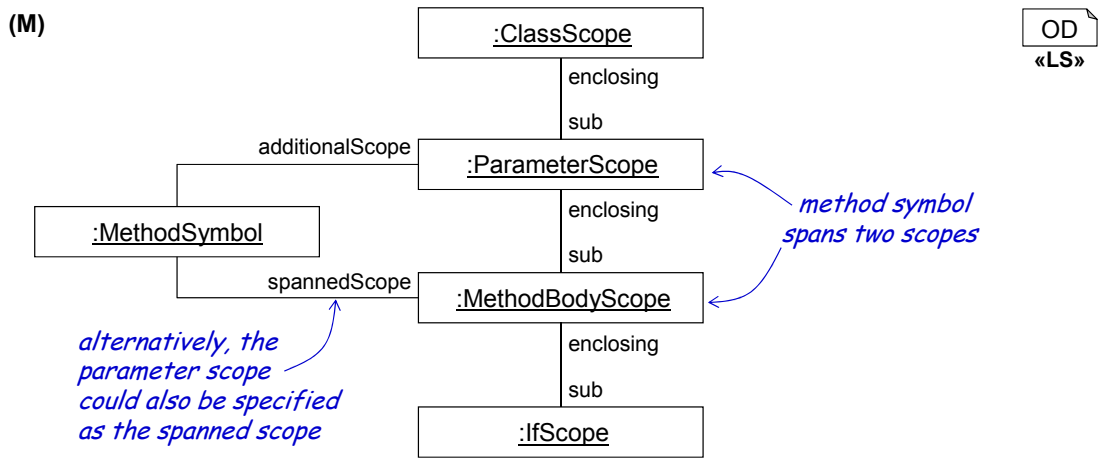


Figure 4.28: General idea of the pattern *(M) Method Spanning a Parameter Scope and a Body Scope.*

the parameter scope and the method body scope are logically treated as a single scope since the search is always conducted in both scopes.

(N) Method Spanning an Intermediate Method Scope

Same as before, the second pattern separates parameter scope and method scope but additionally introduces an intermediate scope to group these two. As it can be seen in Figure 4.29, the method symbol spans only one scope in this pattern, i.e., the method scope. The method scope serves as intermediate scope and groups the parameter scope and the method body scope. Hence, the distinction between formal parameters and local variables as in pattern *(M) Method Spanning a Parameter Scope and a Body Scope* still exists. The parameter scope and the method body scope are now encapsulated in the method scope. Listing 4.30 demonstrates the implementation of the methods `getParameters` and `getVariables` of the `MethodSymbol` class. As it can be seen, the former delegates to the parameter scope via its spanned scope while the latter delegates to the method body scope[6].

With this pattern the scope structure becomes more complex since it requires three scope classes. Consequently, same as the pattern *(M) Method Spanning a Parameter Scope and a Body Scope*, it is not very intuitive. Moreover, the language engineer must determine which scope ultimately is the subscope of `MethodScope`. In Figure 4.29 the subscope is the if scope. Alternatively, the if scope can be considered as a subscope of `MethodBodyScope` which in turn is a subscope of `MethodScope`. However, none of these alternatives preserve the scope tree structure discussed below.

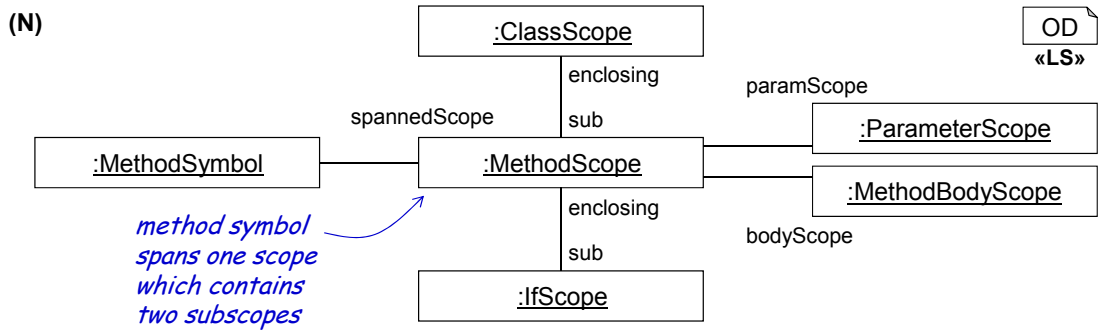


Figure 4.29: General idea of the pattern *(N) Method Spanning an Intermediate Method Scope*.

[6]Please note that this example assumes the pattern *(D) Same Symbol Class for Similar Model Elements* (cf. Section 4.1.2), i.e., `JavaFieldSymbol` represents parameters, local variables, and fields. Since the symbols are stored in different scopes, there is no need for filtering out any symbols.

```java
public Collection<JavaParamSymbol> getParamters() {
  return getSpannedScope()
    .getParamScope().resolveLocally(JavaFieldSymbol.KIND);
}

public Collection<JavaVariableSymbol> getVariables() {
  return getSpannedScope()
    .getBodyScope().resolveLocally(JavaFieldSymbol.KIND);
}
```

Java «LS»

Listing 4.30: Implementation of the pattern *(N) Method Spanning an Intermediate Method Scope*.

(O) Method Spanning Only a Method Scope

Finally, the last pattern leads to the least complex scope structure. As presented in Figure 4.31, there is no more distinction between a scope for formal parameter definitions and local variable definitions in a method. The method scope unifies a method's parameter scope as well as its body scope, following Gabbrielli et al. that a "[b]lock associated with a procedure [...] corresponds [...] to the body of the procedure itself, extended with the declarations of formal parameters" [GM10].

One advantage is the simple and intuitive scope structure that emerges. Also, this conforms to Java's or C#'s handling of formal parameters and local variables. First, both can only be used within the method body [GJS+14]. Hence, they are *visible within the same scope* (cf. Def. 3.6). Second, a method's formal parameters and its local variables may not be same-named, which indicates that they are *defined in the same scope*.

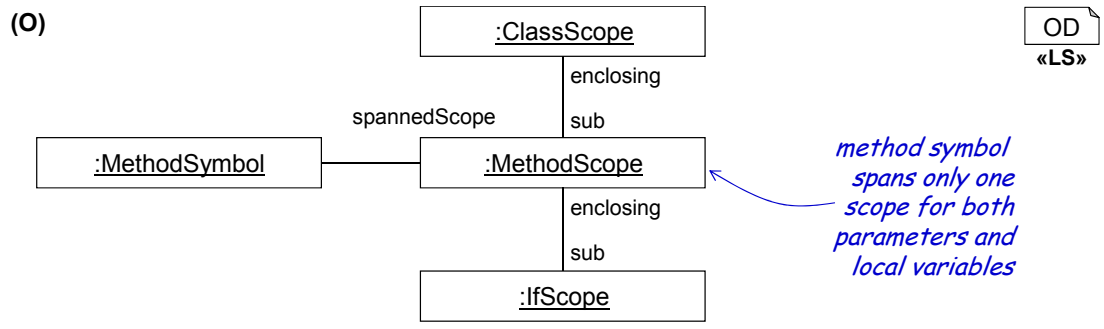


Figure 4.31: General idea of the pattern *(O) Method Spanning Only a Method Scope*.

Consequently, the only reason for separating parameter scope and method body scope is to distinguish between parameter symbols and local variable symbols. This, however, can be solved via different symbols (i.e., pattern *(E) Different Symbol Classes for Similar Model Elements*, cf. Section 4.1.2) and different symbol kinds (i.e., pattern *(G) Different*

Symbol Kinds for Similar Model Elements, cf. Section 4.1.3), as shown in Listing 4.32. Here, `getParameters` delegates to the method symbol's spanned scope in order to locally resolve the parameter symbols (line 3). Analogously, `getVariables` resolves local variable symbols (line 8).

When applying the pattern *(D) Same Symbol Class for Similar Model Elements* (cf. Section 4.1.2), the method symbol can help to filter out unmatched symbols, analogously to Listing 4.24 (on page 78). Alternatively, a resolution via predicates can be conducted (cf. Section 6.7).

Java «LS»

```
1 public Collection<JavaParamSymbol> getParamters() {
2   return
3    getSpannedScope().resolveLocally(JavaParamSymbol.KIND);
4 }
5
6 public Collection<JavaVariableSymbol> getVariables() {
7   return
8    getSpannedScope().resolveLocally(JavaVariableSymbol.KIND);
9 }
```

Listing 4.32: Example of a method symbol that delegates to its spanned scope in order to resolve parameters and local variables. This example assumes patterns *(E) Different Symbol Classes for Similar Model Elements* (cf. Section 4.1.2) and *(G) Different Symbol Kinds for Similar Model Elements* (cf. Section 4.1.3).

Discussion

The first two patterns presented in this section, i.e., *(M) Method Spanning a Parameter Scope and a Body Scope* and *(N) Method Spanning an Intermediate Method Scope* yield the advantage that they preserve the syntactic separation of a method's formal parameters and its local variables by using separated scopes in the scope graph. This further allows for applying pattern *(D) Same Symbol Class for Similar Model Elements* (cf. Section 4.1.2) without additional filtering since the symbols are stored in different scopes.

However, as mentioned above, parameters and local variables have the same visibility (cf. Def. 3.6), i.e., both are visible within the method body. Also, parameters and local variables within the same method may not be same-named. Consequently, they are (logically) defined in the same scope. The third pattern *(O) Method Spanning Only a Method Scope* (cf. Section 4.3.2) best fits this viewpoint, and thus, should be used in order to emphasize this semantics (cf. Def. 3.15). Furthermore, this pattern simplifies both the bottom-up resolution starting with the inner most scope (cf. [ALSU06]) as well as the top-down resolution, as elaborated in Chapter 6. The main drawback of this pattern is that it requires additional filtering if pattern *(D) Same Symbol Class for Similar Model Elements* (cf. Section 4.1.2) is applied (all symbols are stored in method scope).

4.4 Technical Realization of Symbol References

A symbol reference—as introduced in Section 3.4—is represented by the interface `SymbolReference` depicted in Figure 4.33. The generic type argument `T` enables to implement language-specific references, such as references for field symbols and method symbols. `SymbolReference` provides three methods:

isReferencedSymbolLoaded() Checks whether the referenced symbol (i.e., the symbol definition) is already loaded. This information is, among others, employed to load the referenced symbol only once.

getReferencedSymbol() Returns the corresponding symbol definition. If required, the symbol will be loaded first.

existsReferencedSymbol() Checks whether the respective symbol definition exists. For this, it eventually tries to load the symbol first (via `getReferencedSymbol`).

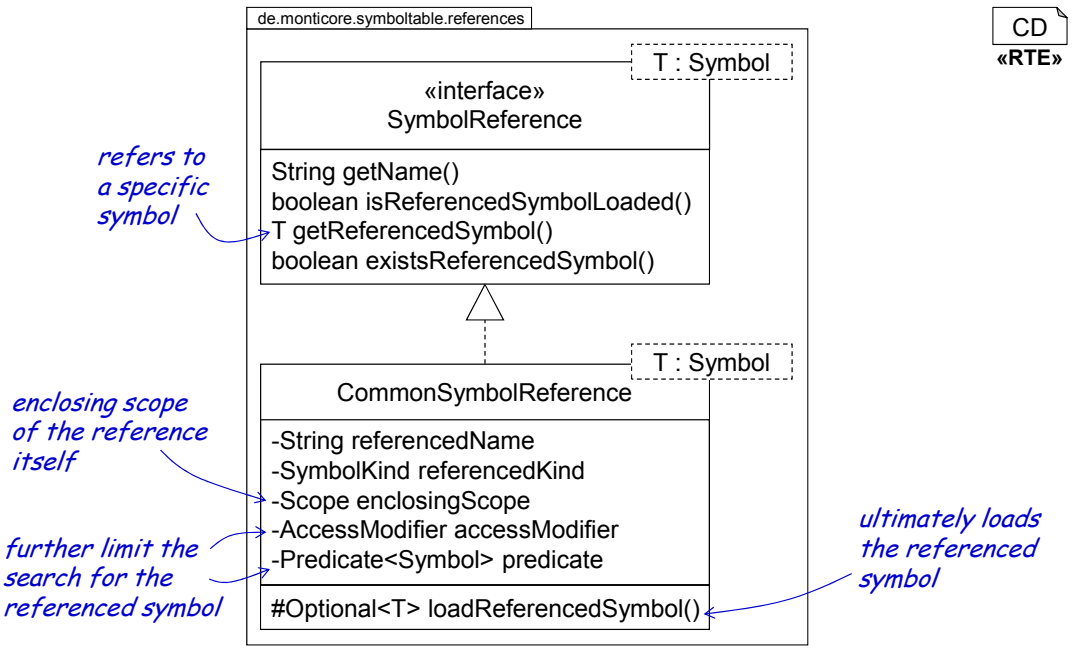


Figure 4.33: Overview of the technical classes for symbol references introduced in Section 3.4. `SymbolReference` is the supertype of all symbol references and `CommonSymbolReference` its default implementation.

`SymbolReference` does not specify how a symbol has to be searched or how it is loaded. In contrast, the default implementations provided by `CommonSymbolReference` highly depend on the symbol resolution and model loading process discussed

in Chapter 6. Figure 4.33 highlights that `CommonSymbolReference` needs further information in order to find the respective symbol definition: the name of the referenced symbol (`referencedName`), its kind (`referencedKind`), and the enclosing scope of the reference (`enclosingScope`). The latter serves as starting point for resolving the symbol definition. Optionally, an access modifier (`accessModifier`) and a predicate (`predicate`) can be passed to further constrain the search.

By default, the symbol resolution only starts on demand (same as [EB10, Völ11]), by invoking `getReferencedSymbol()`. Figure 4.34 illustrates the general procedure. If the symbol definition is already loaded, the symbol will be returned. Else, `loadReferencedSymbol` starts the symbol resolution process by calling the respective `resolve(...)` method of the enclosing scope (cf. Section 6.8).

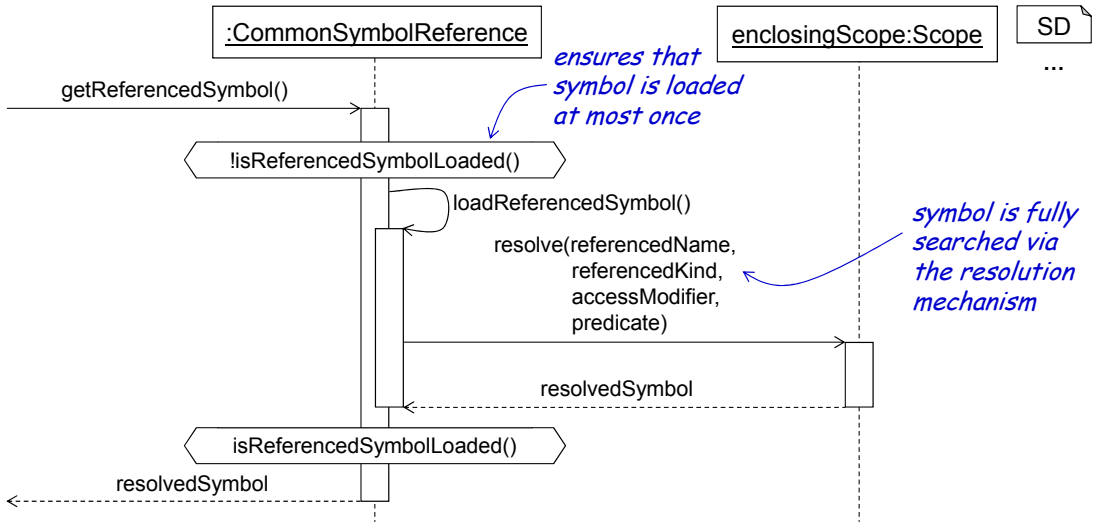


Figure 4.34: Exemplary procedure conducted by `CommonSymbolReference` to (lazily) search for the referenced symbol.

If the symbol still cannot be found (not shown in Figure 4.34), a `FailedLoadingSymbol` exception will be thrown since referential integrity is violated: a symbol that does not exist is referenced. Consequently, the (referencing) model is not well-formed (or the model path is not set correctly, cf. Section 2.2.6). The default implementation provided by `CommonSymbolReference` assumes that the *models are well-formed*, and thus, that the symbol definitions exist. Therefore, the non-existence of the symbol is considered as an exception.

`CommonSymbolReference`'s `existsReferencedSymbol` method serves as *state-testing method* [Blo08] for checking whether the symbol definition exists to avoid that an exception is thrown. In order to determine whether a symbol exists, it has to be loaded first. For this, `existsReferencedSymbol` delegates to `loadReferencedSymbol` (same

as `getReferencedSymbol`), which ultimately starts the resolution process. To adapt the default behavior of both methods `existsReferencedSymbol` and `getReferencedSymbol`, the language engineer only needs to override `loadReferencedSymbol`. The other methods remain unchanged.

Please note that `SymbolReference` presents both a reference within the same model (i.e., intra-model reference) and a reference between different models (inter-model reference), as defined in Def. 3.3. The underlying resolution process automatically determines the respective symbol definition, and thus, simplifies the work of the language engineer.

To sum up, `SymbolReference` and its default implementation `CommonSymbolReference` encapsulate the symbol definition finding process. That way, neither language engineers nor language users need to understand this process in detail. Additionally, the complete resolution and model loading process as described in Chapter 6 are reused which, among others, include the shadowing and visibility rules of the different symbol kinds (cf. Section 3.5). It also considers language composition, which is realized based on the resolution process (cf. Chapter 8).

4.4.1 Patterns for Symbol References

The remainder of this section elaborates three patterns for implementing symbol references by the example of Java type symbols.

(P) Symbol Reference Using Delegation

A symbol reference can be realized using delegation (cf. [HMSNR15b]). In Figure 4.35 `JavaTypeSymbolReference` represents the reference of a `JavaTypeSymbol`. For this, it implements `SymbolReference`. To reuse the default behavior implemented in `CommonSymbolReference`, `JavaTypeSymbolReference` can either extend `CommonSymbolReference` (as shown in Figure 4.35) or delegate to it. Furthermore, `JavaTypeSymbolReference` is associated with a `JavaTypeSymbol`. Since the definition is loaded lazily (see above), the cardinality is `0..1`.

This approach has the advantage that the symbol definition and its reference(s) are separated. That way, reference-specific information, e.g., type arguments, can be stored in `JavaTypeSymbolReference` (cf. Section 3.4). Also, it enables to explicitly distinguish between a definition and a reference by using the corresponding type in the source code. The main disadvantage of this pattern is that the language user always must be aware of whether to employ a symbol definition or a reference even if a distinction is not useful (see next pattern).

(Q) Symbol Reference Using Proxy Pattern

A special case of delegation is applied in the Proxy pattern; a proxy "provides an interface identical to [s]ubject's so that a *proxy can b[e] substituted for the real subject*" [GHJV95].

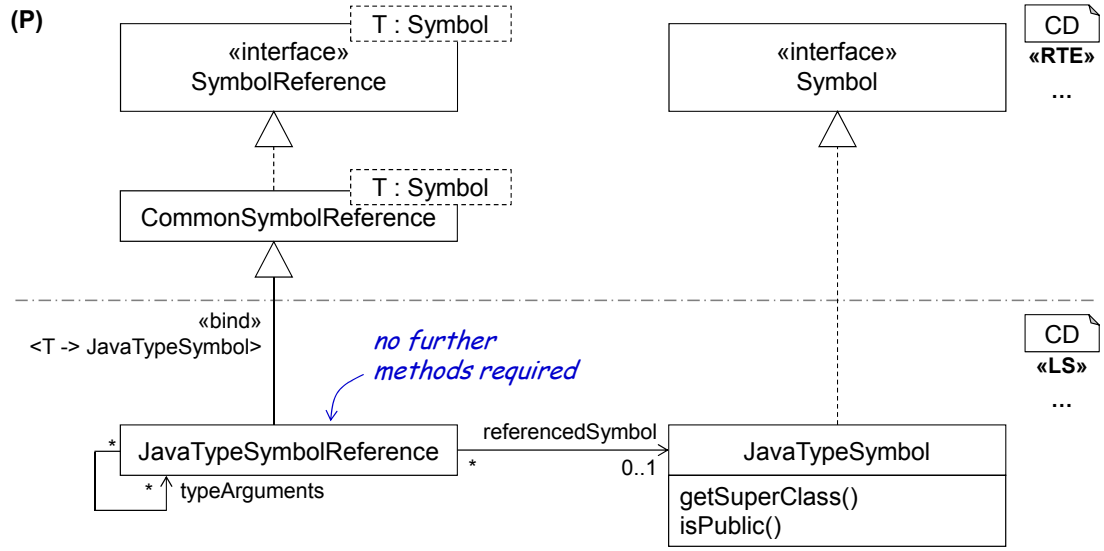


Figure 4.35: General idea of the pattern *(P) Symbol Reference Using Delegation*. Here, information specific to a symbol definition and reference-specific information are strictly separated.

Figure 4.36 shows an example. `JavaTypeSymbol` is the *real symbol* and `JavaTypeSymbolReference` its *proxy*. Since `JavaTypeSymbolReference` subclasses `JavaTypeSymbol`, its interface is identical, thus, it can be substituted for `JavaTypeSymbol`. Moreover, `JavaTypeSymbolReference` refers to `JavaTypeSymbol` in order to *forward every request* to the real symbol (cf. [GHJV95]), i.e., the symbol definition [HMSNR15b]. The symbol reference itself solely contains the required information—i.e., at least the name and kind of the referenced symbol—for resolving the corresponding definition. Furthermore, `JavaTypeSymbolReference` delegates to `CommonSymbolReference` to conduct the symbol resolution.

This pattern simplifies the symbol table structure for languages where references do not have additional information. For example, consider a language for automatons which has states and transitions (cf. Section 7.2). A transition has a source and a target state, and hence, refers to two states. By using the proxy pattern for state references, `TransitionSymbol` can be associated directly with `StateSymbol` instead of `StateSymbolReference`. Although technically instances of `StateSymbolReference` are passed to `TransitionSymbol`, this aspect is hidden from the language user, in contrast to the previous pattern. The major drawback of this pattern is that the proxy class must override all methods of the real symbol. This is an error-prone task, especially if the real symbol (i.e., the symbol definition) changes.

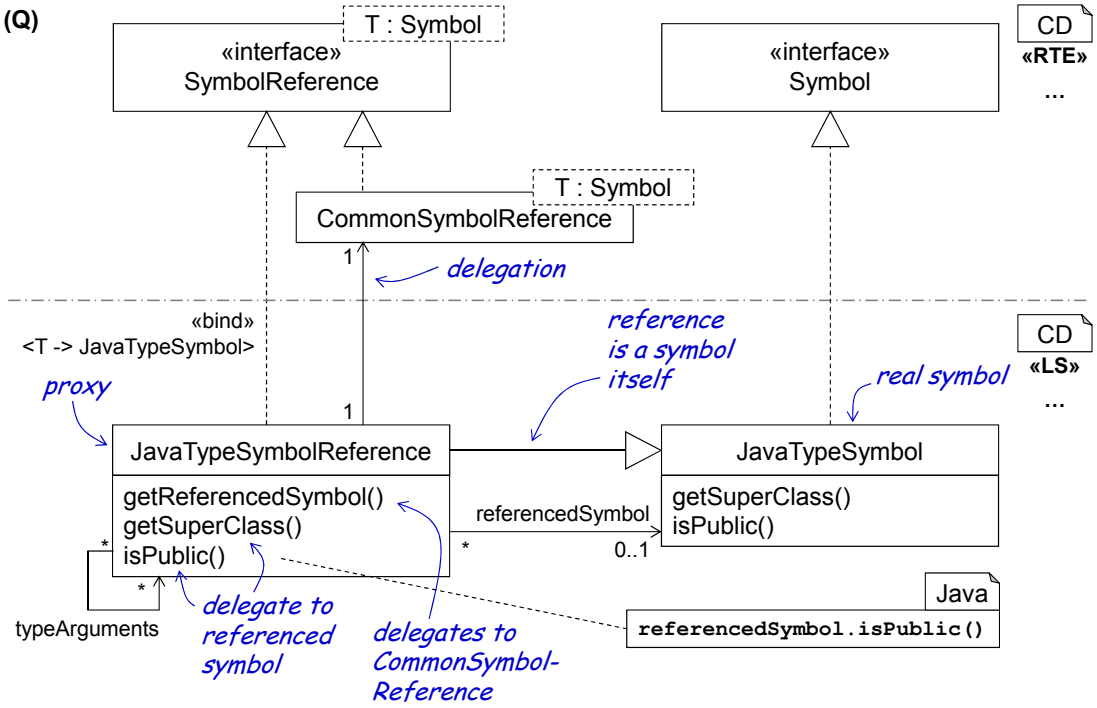


Figure 4.36: General idea of the pattern *(Q) Symbol Reference Using Proxy Pattern* (cf. [GHJV95]). Here, the symbol reference is a symbol itself, which can be substituted for the symbol definition. For this, the symbol reference delegates every request to the definition.

(R) Same Class for a Symbol Definition and Its References

Völkel [Völ11] suggests an approach where a symbol definition and the corresponding symbol references are represented by the same class. Whether the class is a definition or a reference is determined by the symbol table entry states `unqualified`, `qualified`, and `full` (cf. Figure 4.7 on page 57). While references are either `unqualified` or `qualified`, definitions are always `full`. Figure 4.37 shows how the concept of Völkel can be applied to the current infrastructure.

`JavaTypeSymbol` represents both the definition and the reference of a Java type, determined by the method `isReference`. Reference-specific information (such as type arguments) is directly bound to `JavaTypeSymbol`. It is important that both associations `typeArguments` and `referencedSymbol` are only set for references, i.e., in case `isReference() == true`.

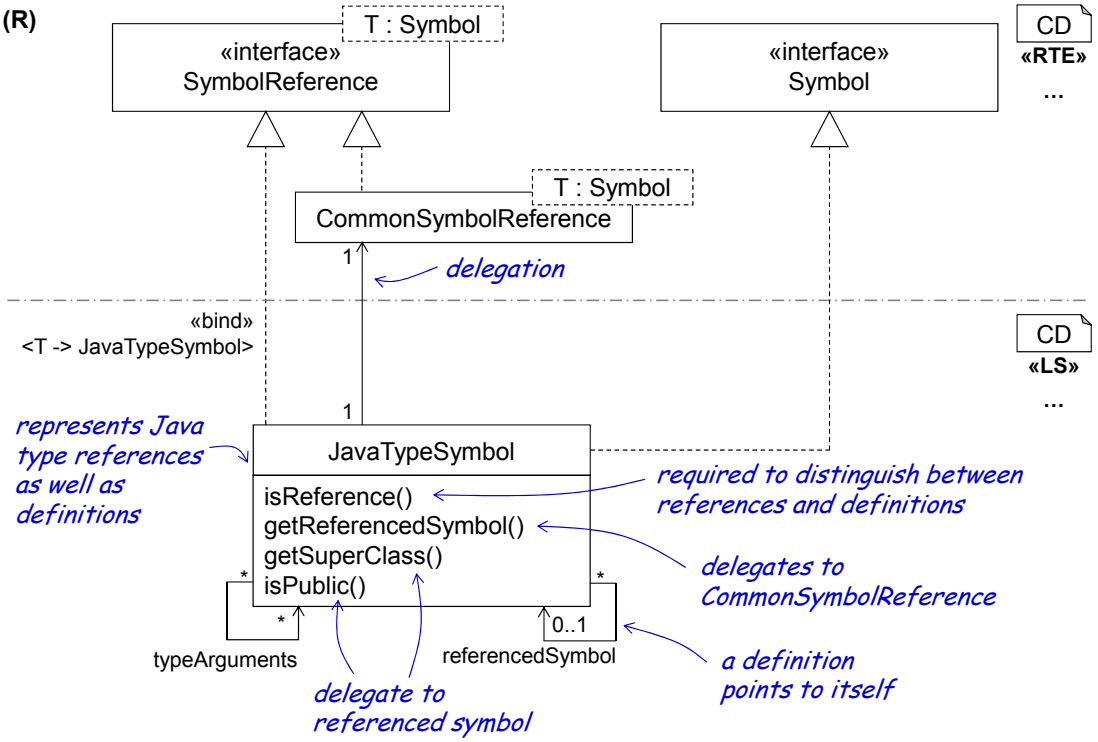


Figure 4.37: General idea of the pattern *(R) Same Class for a Symbol Definition and Its References* (cf. [Völ11]). Here, both the symbol definition and its references are represented by the same class. An additional method (e.g., `isReference()`) is required to distinguish between definitions and references.

Compared to the pattern *(Q) Symbol Reference Using Proxy Pattern*, the current pattern is more robust regarding language evolution since only one class has to be maintained. However, it yields some crucial drawbacks. First, the class `JavaTypeSymbol` is polluted with both symbol definition information and symbol reference information, which leads to the second disadvantage. Each method must distinguish between being called for a definition or for a reference. Listing 4.38 highlights this by the example of `JavaTypeSymbol`'s `getSuperClass` method. In case it is called for a reference (line 2), it delegates to the corresponding symbol definition (line 3). Otherwise, it returns the superclass (line 6).

```java
public Optional<JavaTypeSymbol> getSuperClass() {
  if (isReference()) {
    return getReferencedSymbol().getSuperClass();
  }

  return Optional.ofNullable(superClass);
}
```

Java «LS»

Listing 4.38: Implementation of the pattern *(R) Same Class for a Symbol Definition and Its References* by the example of `JavaTypeSymbol`'s `getSuperClass` method (cf. [Völ11]).

Discussion

The patterns *(P) Symbol Reference Using Delegation* and *(Q) Symbol Reference Using Proxy Pattern* both separate concerns by separating definitions and references into different classes. Consequently, the corresponding information is not intertwined as in pattern *(R) Same Class for a Symbol Definition and Its References*.

Furthermore, the delegation approach (i.e., pattern *(P) Symbol Reference Using Delegation*) enables a clear separation of symbol definitions and references in the source code by using the corresponding class, e.g., `JavaTypeSymbol` or `JavaTypeSymbolReference`. While this is not possible with pattern *(R) Same Class for a Symbol Definition and Its References*, the proxy approach (i.e., pattern *(Q) Symbol Reference Using Proxy Pattern*) enables this to a certain extent; a reference can be enforced by stating its class in the source code, e.g., `JavaTypeSymbolReference`. In contrast, a symbol definition cannot be enforced since a reference class subclasses the definition class, and hence, can be used wherever a definition may be used.

Unlike pattern *(P) Symbol Reference Using Delegation*, the last two patterns yield the advantage that references can be encapsulated, hence, the language user does not have to distinguish them from definitions. This is useful for languages where references do not contain additional information, as in the automaton language presented in Section 7.2.

Finally, in contrast to the other two approaches, the pattern *(Q) Symbol Reference Using Proxy Pattern* is based on inheritance (i.e., reference class extends symbol class), and thus, might lead to additional effort when inheriting the language (cf. Section 8.4) since Java only enables single inheritance.

To sum up, in general pattern *(P) Symbol Reference Using Delegation* is recommended since it not only separates references and definitions explicitly but also is robust against language changes that only concern the symbol definition. However, if the reference should be abstracted away, pattern *(Q) Symbol Reference Using Proxy Pattern* is to be favored over pattern *(R) Same Class for a Symbol Definition and Its References* since the former does not intertwine reference and definition implementations. The latter, however, facilitates maintenance as only one class exists.

The Xtext [Bet13] language workbench employs a combination of pattern *(P) Symbol Reference Using Delegation* and pattern *(Q) Symbol Reference Using Proxy Pattern* for lazy linking. Based on EMF [SBPM09] dedicated classes exist for references and definitions. However, a reference is not a proxy itself but uses one that ultimately delegates to the real object.

4.5 Technical Realization of Access Control Mechanisms

Access modifiers—as defined in Def. 3.13—are top-level concepts in SMI, represented by a dedicated interface. Figure 4.39 depicts the generic type hierarchy of access modifiers.

The interface `AccessModifier` is the root type of all access modifiers. Its `includes` method determines whether the inclusion relation between two access modifiers exists, as defined in Def. 3.14. The constant `ALL_INCLUSION` is an access modifier that includes any other access modifier, and thus, enables to resolve symbols regardless of their specified access modifier (cf. Chapter 6).

Following the Java language, `BasicAccessModifier` defines four access modifiers, namely public, protected, package-local, and private. Listing 4.40 demonstrates the `includes` method for the package-local access modifier. As it can be seen, the method solely checks whether the passed access modifier is either `PUBLIC`, `PROTECTED`, or `PACKAGE_LOCAL`. Hence, a symbol must be at least package-local in order to be found.

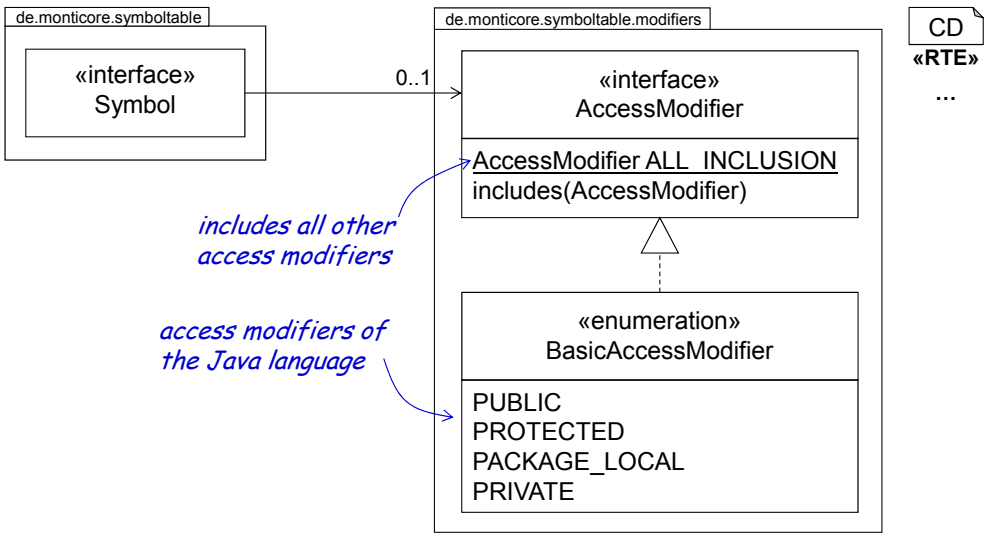


Figure 4.39: Overview of the technical classes for access modifiers as introduced in Section 3.7. The interface `AccessModifier` is the supertype of all access modifiers.

```java
@Override
public boolean includes(AccessModifier modifier) {
  return (modifier.equals(PUBLIC)
      || modifier.equals(PROTECTED)
      || modifier.equals(PACKAGE_LOCAL));
}
```

Java «RTE»

Listing 4.40: Implementation of the `includes` method for the package-local access modifier of `BasicAccessModifier`.

An enum is the natural form of implementing access modifiers since they are fixed within a specific language. However, if language-specific access modifiers are required and explicitly stated in the source code (e.g., `JavaAccessModifier` instead of `AccessModifier`), using a dedicated class for each modifier can be more suited (e.g., `JavaPublicModifier`, `JavaProtectedModifier`, etc.). This approach simplifies language inheritance, where the extending language can add new modifiers via subclassing.

Access modifiers are essential for exporting and importing symbols (cf. Section 3.10). In Java, for example, a class scope only exports fields and methods that are not private. Hence, a class scope can only import non-private members of its superclass' scope. While this thesis employs access modifiers as top-level concepts, Völkel [Völ11] instead provides different kinds of symbol tables, as already discussed in Section 3.10. The top part of Figure 4.41 shows the symbol table architecture as described in [Völ11]. As it can be seen, it differs from the symbol table architecture of SMI (bottom part of Figure 4.41) in many ways. Besides the different terminology (i.e., `NameSpace` and `Entry` instead of `Scope` and `Symbol`), the entries are not directly contained in the namespaces. Instead, each namespace can contain up to four different symbol table kinds (namely, encapsulated, imported, exported, and forwarded, cf. Section 3.10) that in turn contain the entries.

As shown in the bottom part of Figure 4.41, the symbol table infrastructure of SMI does not provide different types of symbol tables. Instead, each symbol optionally has a modifier. Moreover, it depends on a scope whether its symbols are exported or not, stated by the `exportsSymbols` method (cf. Section 4.2). The resolution mechanism explicitly considers modifiers (together with their inclusion relation) as well as the exporting ability of scopes which yields the following advantages over Völkel's [Völ11] approach:

- It simplifies the infrastructure since solely one scope is required to manage the symbols having different modifiers. In contrast, the infrastructure in [Völ11] requires a symbol table kind per modifier type.
- Völkel [Völ11] does not explicitly provide the inclusion relations of modifiers, and hence, their implementation is more cumbersome then the `includes` method shown in Listing 4.40. Technically, this means, for example, that all entries of the *public* symbol table kind must also be imported into the other symbol table kinds.

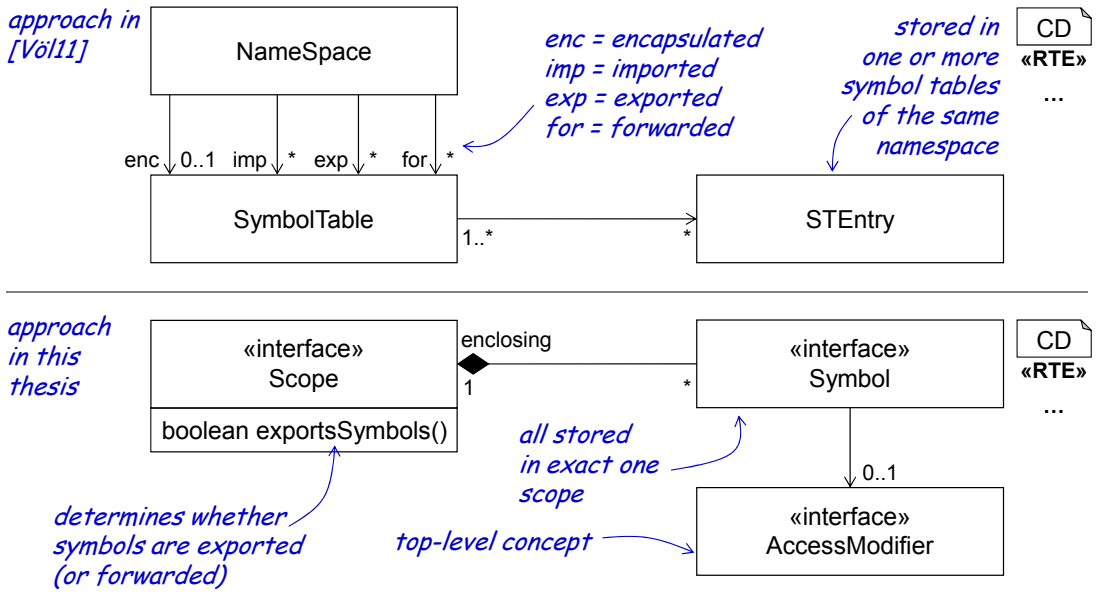


Figure 4.41: Comparison of Völkel's [Völ11] symbol table kinds (top part) and SMI's access modifiers (bottom part).

- The symbol table engineer is liberated from working with different kinds of symbol tables, especially during their creation. Section 5.6 discusses this topic.
- It is ensured that all symbols in a scope behave the same regarding their resolution (cf. Section 3.5). For example, while (non-private) inner classes in Java are exported (same as fields and methods), classes defined in methods are not (same as formal parameters and local variables). This behavior is also guaranteed for symbols that are embedded in scopes of other languages (cf. Section 8.2).

4.6 Technical Relation of AST and Symbol Table

The AST is an internal representation of the (parsed) model. The symbol table—with its scopes and symbols—usually is an abstraction of the model, and hence, of the AST (cf. Section 3.8). Depending on the task to be conducted (e.g., code generation or context conditions checking), either or both are required [HMSNR15b]. For this reason, SMI enables linking AST nodes with corresponding symbols and scopes, as depicted in Figure 4.42.

The association between `ASTNode` and `Symbol` occurs in case they represent the same model element, for example, as the classes `ASTJavaField` and `JavaFieldSymbol` both represent a Java field. The association is optional since an AST node does not

necessarily have a corresponding symbol as, for example, `ASTImportStatement`, which represents an import statement. Furthermore, the symbol table is built *after* the model is parsed (cf. Chapter 5). Hence, the AST nodes are created before their corresponding symbols. Analogously, it is also possible to create a symbol (or parts of it) independently of an AST node (cf. Section 3.8).

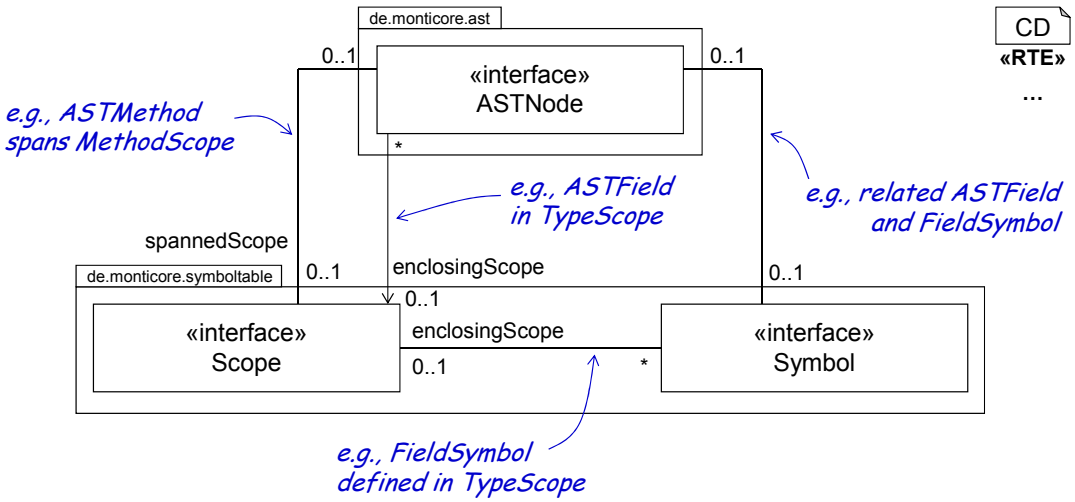


Figure 4.42: Relation between `ASTNode`, `Symbol`, and `Scope`.

In case an AST node is represented by more than one symbol, it can be extended with a *sym*`Symbol` field (with getters and setters) for each symbol `sym` (using MontiCore's extension mechanism, cf. Section 7.14). For example, a bidirectional association in a class diagram can be represented by two symbols: one symbol for the left-to-right association and another symbol for the right-to-left association. For this, the association AST requires the fields `leftToRightSymbol` and `rightToLeftSymbol`. As described above, the relation should be optional. The two association symbols are linked to the (same) AST node via the generic relation shown in Figure 4.42, since each is related to (at most) one AST node.

Some AST nodes represent model elements that span a scope (e.g., a Java method), and hence, are linked to the corresponding spanned scope (and vice versa). This relation is optional since not all model elements span a scope (e.g., a Java field), and not all scopes are spanned by a model element (e.g., the global scope).

The enclosing scope of an AST node is the scope spanned by its parent node or—if it is the root node—the global scope. Same as before, the relation is optional since the two structures can exist independently.

According to Völkel [Völ11], the link between an AST node and the corresponding

symbol (table entry) should be omitted since it can, among others, hamper language composition. To avoid this disadvantage but still enable easy access to both structures, their relation in SMI is *optional*, hence, the language user must explicitly be aware that the relation might be missing. Furthermore, the relation is *language-unspecific*, i.e., the symbol does not know the explicit type of its corresponding AST node (and vice versa). Otherwise, not only the symbols would require adaption when composing languages but also the AST nodes, as already discussed for pattern *(A) Symbol Provides No Information Directly Contained in Related AST Node* (cf. Section 4.1.1). Similar to the relation between AST nodes and scopes in our approach, Völkel [Völ11] associates AST nodes to namespaces (called "associated namespaces") and vice versa. Parr [Par10] links AST nodes to their scope but omits the opposite direction. Same as in our approach, AST nodes and symbols are linked together in order to get information of either one. Frameworks based on EMF [SBPM09] (e.g., EMFText [HJK+09] and Xtext [Bet13]) rely on the Ecore meta-model which specifies the abstract syntax, and thus, no structures need to be linked. Similarly, frameworks applying a projectional approach, such as MPS [VS10], focus on the AST only. In contrast, in the current thesis, the abstract syntax consists of both the AST and the symbol table (cf. Section 3.8) which therefore are linked.

Attribute grammars [Knu68] allow to declaratively specify attributes for productions of formal grammars. The value of an attribute is associated with the respective node of the AST tree and computed in one or more traversals of that tree. Many approaches employ and extend the classical attribute grammars in order to enable a modular definition of a language (e.g., [Hed89, VSK89, DC90, FMY92, KW94, MLAŽ99, WMBK02]). Reference attributed grammars (RAGs) [Hed00] extend the original attribute grammars with references between AST nodes and are employed in tools, such as Silver [WBGK10], Kiama [SKV10], ASTER [KSV09], and JastAdd [HM03]. In the following we only discuss JastAdd since the main ideas among those tools are similar [Hed11].

JastAdd [HM03, EH07] is a meta-compilation system for generating language-based processors such as compilers and source-code analyzers. It combines RAGs and object-orientation, in particular, Java. Furthermore, JastAdd enables a modular specification of different aspects of the language, e.g., name analysis and data flow analysis in an aspect-oriented manner. Besides simple values (e.g., integer) and composite values (e.g., collection), JastAdd allows for reference values linking to other AST nodes. That way the syntax tree becomes an object-oriented graph model [Hed11]. This is similar to our approach where the scope graph enriches the AST with, among others, references. However, JastAdd embeds the symbol table in the AST [Hed11] while in the current thesis the two structures are (optionally) linked together and can be used independently from each other. This, above all, allows for defining symbol tables that essentially differ from the AST structure which is not possible in JastAdd since the AST itself is extended.

4.7 Naming Conventions

In order to improve the implementation's readability, language-specific symbol table classes should adhere to the naming conventions listed in Table 4.43. In short, each language-specific class name should be suffixed with the component's name it implements, e.g., `JavaField`**`Symbol`**. As also indicated in Table 4.43, a prefix of the language (e.g., "Java") can help to distinguish between similar components of different languages, for example, **`Java`**`FieldSymbol` and **`CSharp`**`FieldSymbol`.

SMI Component	Convention	Example
Symbol	suffix "Symbol"	`JavaFieldSymbol`
SymbolKind	suffix "Kind" or suffix "SymbolKind"	`JavaFieldKind` or `JavaFieldSymbolKind`
Scope	suffix "Scope"	`JavaTypeScope`
ScopeSpanningSymbol	suffix "Symbol"	`JavaTypeSymbol`
SymbolReference	suffix "Reference" or suffix "SymbolReference"	`JavaTypeReference` or `JavaTypeSymbolReference`

Table 4.43: Naming conventions for language-specific implementations of `Symbol`, `SymbolKind`, `Scope`, `ScopeSpanningSymbol`, and `SymbolReference`.

Chapter 5

Building Up Language-Specific Symbol Tables: Method and Implementation

Chapter 4 introduces the generic and reusable infrastructure SMI that serves as a basis for implementing language-specific symbol tables. Given that the data structures (classes) for the symbol table are defined, in this chapter we discuss how to build the symbol table (objects) for a model. This is because, before the symbol table of a model[1] can be used (e.g., to check context conditions) it has to be created first. In classical compiler construction, this is often conducted by the *parser* during the analysis phase [ALSU06, Seb08]. In the current thesis the symbol table creation is explicitly separated from the parsing process. It is rather based on the AST, and hence, has to be created *after* the parsing process. This yields the following benefits:

- It enables to conduct transformations [MCG05] on the AST, e.g., by simplifying it (cf. "desugaring" [KV10]), *after* the parsing process. The symbol table then can be created based on the *transformed* AST. That way, consistency between these two structures is ensured by construction.
- Furthermore, the AST is not necessarily created during the parsing process. For instance, AST providers (cf. Section 6.9.3) hide the AST creation phase, which is important for integrating the symbol table into an IDE. ASTs created this way can also serve as input for the symbol table creation.

Figure 5.1 gives an overview of the symbol table creation in this thesis. Given a model's AST, the symbol table creator (i) builds up a respective scope graph and (ii) links it with the corresponding AST nodes. That way, the symbol table, among others, enables navigating between the AST nodes (cf. [KRV07b]).

Chapter Outline

The remainder of this chapter is structured as follows. Section 5.1 presents SMI's phases for building a model's symbol table from an input AST and compares it with related

[1] The term "symbol table of a model" refers to the corresponding scope (sub)graph of that model (cf. Def. 3.15).

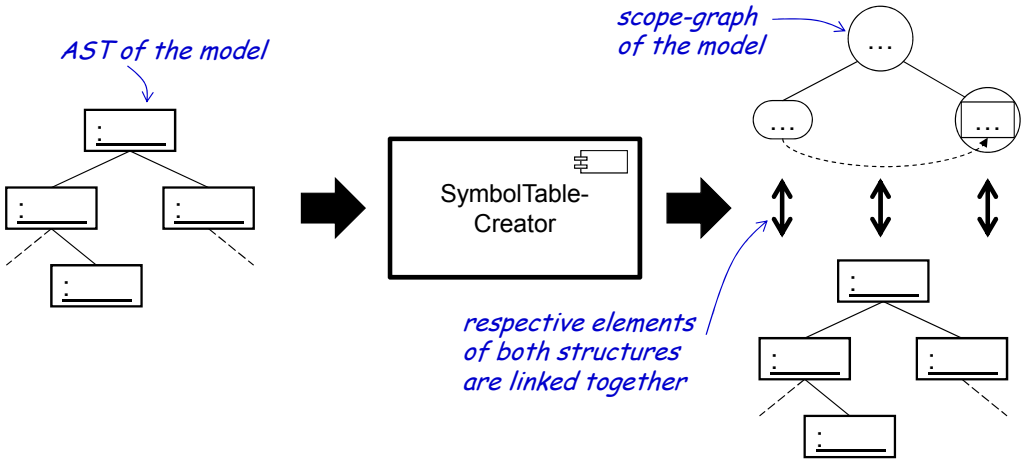


Figure 5.1: Overview of the symbol table creation.

approaches. Next, Section 5.2 gives a method for conducting the presented phases on model elements, depending on their relation to symbol table elements. Subsequently, Section 5.3 demonstrates a top-down depth-first approach for realizing the symbol table creation based on the Visitor pattern [GHJV95] and gives a method for it. In Section 5.4, the language-specific as well as language-unspecific steps for linking AST nodes and symbol table elements are discussed. Afterwards, Section 5.5 demonstrates the implementation of a symbol table creator by the example of a simplified AST structure for Java, following the previous introduced methods. Finally, Section 5.6 compares the presented approach with the symbol table creation in previous MontiCore versions as suggested in [Völ11].

5.1 Symbol Table Creation Phases

The symbol table creation consists of the following phases:

P1. Building-up the scope structure In this phase the scope graph of the model is created (cf. [Völ11]). Usually, it is a scope tree since imported scopes are (logically) linked in the last phase **P5**.

P2. Creating symbols and adding them to the enclosing scope In this phase the symbols are created, initialized, and added to the respective enclosing scope.

P3. Creating symbol references and adding them to the referencing symbol Symbol references (cf. Def. 3.3) are created in this phase. Furthermore, they are added to the referencing symbol (not to the enclosing scope).

P4. Linking the AST and the symbol table structures This phase relates symbols and scopes to the respective AST nodes (and vice versa) which allows to switch between both structures as needed [HMSNR15b] (cf. Section 4.6).

P5. Loading referenced symbols and models In this final step the referenced symbols and models are loaded (cf. [Völ11]). Strictly speaking, this phase is not part of the symbol table creation as it is conducted lazily (cf. Section 6.9).

It is essential to conduct the last phase separately, after all model elements are processed. Otherwise, cyclic dependencies between models can prevent the creation process from terminating. Moreover, forward references will not be resolvable at that time if not all model elements are processed yet.

Although the phases can be performed successively, especially the first four phases are usually intertwined in the implementation. The main reason is that they are processed per model element (i.e., AST node), and thus, rely on model-specific information. When conducting the phases sequentially, this information is (repetitively) required through all the phases. This means, the AST or parts of it must be traversed for every of these phases again. In contrast, conducting the phases iteratively as described in the Subsections 5.2.2 and 5.2.3, has the advantage that the AST only needs to be traversed once.

In [Völ11], the symbol table creation consists of overall nine steps. Many of the steps are conducted sequentially by so-called *workflows* [GKR+08, Kra10]. The steps three (*linking namespaces of the model*), four (*qualifying references entries*), six (*importing symbol tables*), and seven (same as third step) are especially necessary for resolving the symbol table entries. In SMI this steps are performed dynamically during the resolving process (cf. Chapter 6), enabled by the modular and functional architecture of MontiCore 4.

Similar to our approach, Parr [Par10] separates the definition from the resolution phase to, among others, enable forward referencing. The former consists of phases **P1**, **P2**, and **P4**. The latter resolves the symbol references eagerly, in contrast to **P5** of the current thesis, where the references are resolved lazily. Moreover, Parr does not use any reference or proxy classes (cf. Section 4.4) but directly links a name to the respective symbol definition. Consequently, no **P3** as in the current thesis exists.

The language workbench Xtext [EB10] enriches the AST with proxies for cross references. Same as phase **P5** those proxies load the respective definition lazily. However, in contrast to our approach, the proxies are set during the parsing process.

Spoofax [KV10] provides the meta-language NaBL [KKWV13] in order to specify name binding rules. The symbol table (or semantic index [KKWV13]) is populated in three phases, namely an annotation phase, a definition site analysis phase, and a use site analysis phase. The first phase annotates definitions and uses (i.e., references) with unique (qualified) URIs which include the namespace (similar to symbol kind in our approach, cf. Section 3.3) as well as the scopes. This phase corresponds to **P1** and partially to **P2** and **P3**. The definition site analysis phase traverses the AST a second

time and stores—similar to **P2**—information associated with definitions in the symbol table based on the URIs determined in the previous phase. Same as **P5**, the use site analysis phase resolves the references and stores the information in the symbol table.

In [SBHWP16] different meta-models for the concrete and the abstract syntax exist. A transformation language is provided to map concepts of the former to concepts of the latter, which then is employed for model processing. The transformation language further allows for specifying name resolution rules in a NaBL-like way. The symbol table creation in this thesis can be considered as a transformation as well, i.e., from the AST to the symbol table. Since both structures are part of the abstract syntax (cf. Section 3.8), the transformation is not from concrete to abstract syntax as in [SBHWP16]. Thus, the AST and the symbol table can be employed (together) for model processing.

5.2 Method for Processing Model Elements During the Symbol Table Creation

Since the AST represents the (parsed) model, it (in many cases) serves as input for creating the symbol table for that model (cf. [Völ11, NTVW15]). Following from Section 4.6 (i.e., *Technical Relation of AST and Symbol Table*), one of the four cases is true when processing a model element (i.e., its AST node):

(a) The model element neither spans a scope nor is represented by a symbol.

(b) The model element spans a scope but is not represented by a symbol.

(c) The model element does not span a scope but is represented by a symbol.

(d) The model spans a scope and is also represented by a symbol, i.e., it is represented by a scope spanning symbol.

The following four subsections demonstrate the processing steps for each of the above cases during the ST creation. They aim at aiding the symbol table engineer in answering the question: *Given the AST node of a model element, which of the ST phases **P1-P4** should be performed on it?*

Phase **P5** is omitted in the following method since it is conducted lazily *after* the model's symbol table is completely built, and not when processing specific model elements.

5.2.1 Method for Processing a Model Element Not Represented by a Symbol Table Element

Since model elements of case **(a)** are neither represented by a scope nor by a symbol, their corresponding AST nodes are not handled *explicitly* during ST creation, but rather are processed in one of the three other cases. If such a model element does not contain

any relevant information for the symbol table, its AST node will be ignored. A typical example are expressions in statements like `int i = 3 + x`. While variable `int i` can be represented by a corresponding symbol (cf. Section 5.2.3), its value, i.e., the AST node of expression `3 + x` is not represented in the symbol table (cf. Section 3.11).

Furthermore, model elements of case **(a)** can be used for initializing parts of the symbol table, e.g., in the *initialize symbol* phase of Figure 5.3. A modifier in Java, for example, is not represented by its own symbol, in contrast to the corresponding class (cf. Section 5.2.3). Hence, the modifier is set when processing the class (not the modifier).

5.2.2 Method for Processing a Model Element Represented by a Scope

Figure 5.2 depicts a method for processing the AST node of a model element that spans a scope (case **(b)**). First, a corresponding scope is created, which then is optionally initialized with further information (phase **P1**). Afterwards, the scope is added to its enclosing scope (if exists), i.e., the enclosing-sub relation is set, as described in Section 4.2.1. Finally, the scope and the AST node are linked (phase **P4**) .

Case **(b)**, among others, applies to the root node of a model. In many textual languages that node spans an artifact scope (cf. Def. 3.10), but does not represent a model element, and hence, is not represented by its own symbol. The artifact scope is initialized with the package information and the import statements used in the model (*"initialize scope"* activity, Figure 5.2). Its enclosing scope is the global scope (*"set enclosing-sub scope relation"* activity). Another example of case **(b)** is a Java if block, for which solely a visibility scope is created.

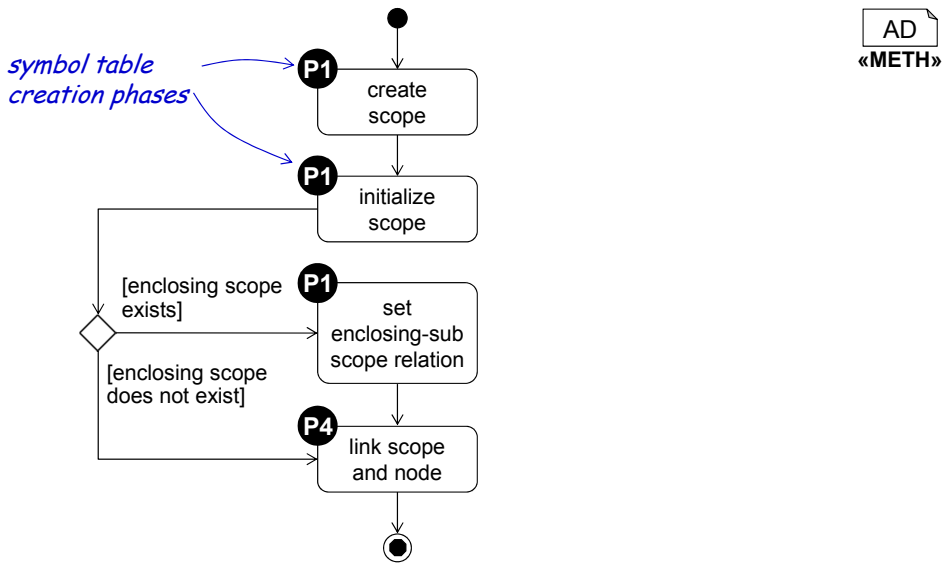


Figure 5.2: Method for processing a model element that spans a scope.

5.2.3 Method for Processing a Model Element Represented by a Symbol

Figure 5.3 presents the steps for processing the AST node of a model element that is represented by a symbol (i.e., case **(c)**). The process is analogous to the one of case **(b)**. First, a corresponding symbol is created and optionally initialized (phase **P2**), which also includes creating symbol references and adding them to the symbol (phase **P3**). Next, the symbol is added to its enclosing scope (if exists), which has been already created during the process of either case **(b)** (cf. Section 5.2.2) or case **(d)** (cf. Section 5.2.4). Finally, the AST node and the corresponding symbol are linked (phase **P4**).

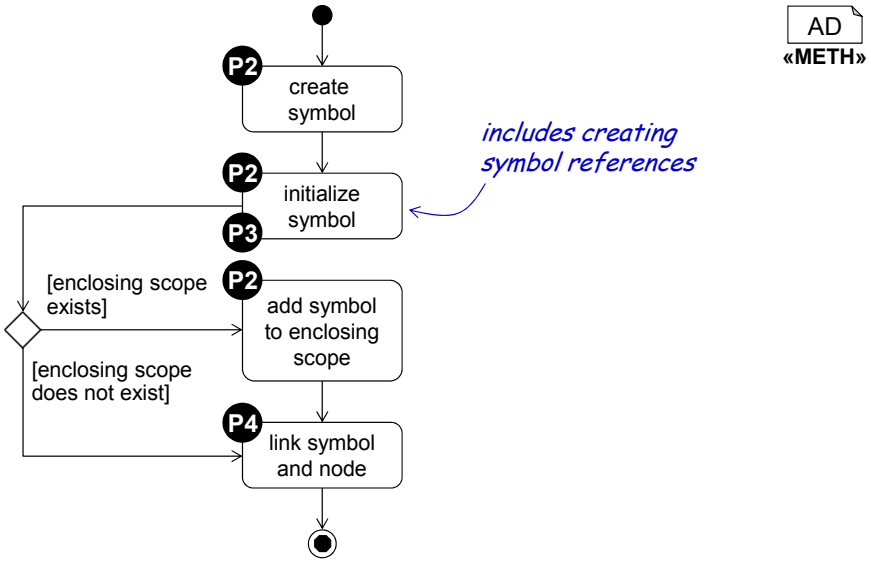


Figure 5.3: Method for processing a model element that is represented by a symbol.

5.2.4 Method for Processing a Model Element Represented by a Symbol and a Scope

The last case **(d)** (i.e., the model element is represented by a scope spanning symbol) is a combination of the two cases **(b)** and **(c)**. The process is depicted in Figure 5.4. After a corresponding symbol and its spanned scope are created, they are linked. In short, the first three steps serve to create a scope spanning symbol and its spanned scope. The remaining steps are the same as in Figure 5.2 and Figure 5.3, respectively.

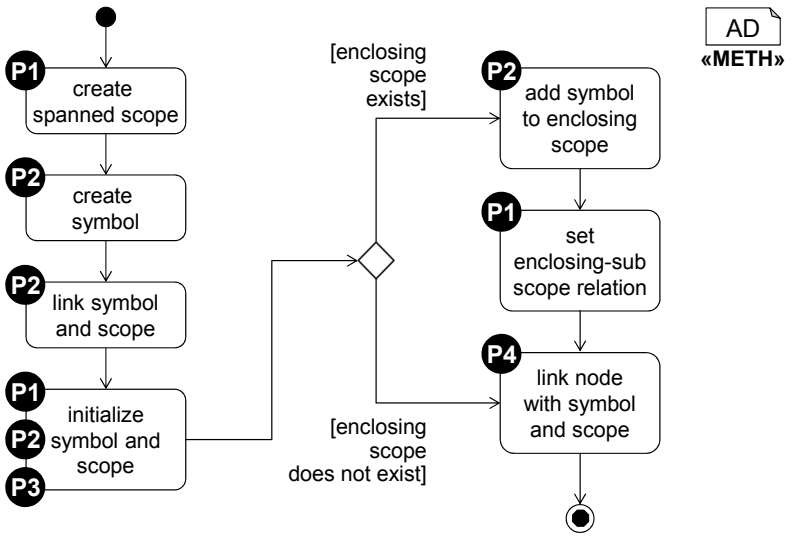


Figure 5.4: Method for processing a model element that is represented by a symbol and a scope.

5.3 Incremental Creation of a Symbol Table using a Depth-First Approach

The symbol table can be created incrementally by traversing the AST using a top-down depth-first approach [ALSU06]. Thus, the parent node is processed *before* its children. Consequently, an enclosing scope is created before its subscopes and containing symbols. This enables to access symbol table information up to the current model element. For example, when processing a method, its enclosing class has already been processed, and hence, the corresponding scope can be accessed. The top-down approach allows (at least) two possible ways for setting the enclosing-sub relations:

- One possibility is to set the relation in the initialization phase of the *enclosing scope*. This entails that the child node is also processed in this phase.
- The other option is to process parent and child node independently from each other. That means, after the processing of the parent node has finished, the processing of the child begins. Consequently, the enclosing-sub relation must be set when the *child node* is processed.

The second case should be favored since it separates the processing of nodes which simplifies reuse and customization for, among others, language inheritance. Furthermore, the processing of the child node does not depend on the parent node anymore. This is especially important for language embedding, where the child node can be reused without its parent node. Hence, the remainder of this chapter is based on the second approach.

5.3.1 Stack-based Approach for Conducting a Top-Down Symbol Table Creation

The scope hierarchy is managed by a stack (cf. [ALSU06, Par10]) where the top element is the current scope and the underlying scope is its direct enclosing scope. When building up a symbol table, such a scope stack is used for an incremental creation of the scope tree. The general process is as follows:

1. Enter scope s: put s on stack. s becomes the current scope. This step occurs only in cases a scope is spanned, i.e., **(b)** and **(d)**.
2. Handle inner elements, e.g., add symbols to current scope.
3. Exit scope s: remove s from stack. Enclosing scope of s (if exists) becomes the current scope. Again, this step occurs only in cases **(b)** and **(d)**.

Please note that while the creation and initialization of symbols and scopes is a *language-specific* task (cf. [NTVW15]), managing the scope stack and adding symbols to the current scope is *language-unspecific*. This enables two important aspects. First, the (generic) stack management can be completely provided by SMI (cf. Section 5.5), and hence, eases reuse for a specific language. Furthermore, it simplifies language embedding. Language embedding is discussed in detail in Section 8.2. Here, it is briefly introduced.

In Java, a method may only be defined in a class. Hence, the enclosing scope of a Java method always is a Java class scope. However, language embedding enables to embed a language or parts of it into another language. So, if a Java method is embedded in a C# class, its enclosing scope will become the C# class. Since adding a symbol to its enclosing scope is language-unspecific, the whole creation process of the Java method can be reused when building the symbol table for the C# class. The remainder of this section demonstrates the stack-based and incremental creation of a symbol table by the example of the Java class presented in Listing 5.5.

Since a top-level Java type is defined in a file, its enclosing scope is an artifact scope (cf. Def. 3.10), which initially is the current scope (step *0.* in Figure 5.6). Class C spans a scope that starts at line 1 and ends in line 6 of Listing 5.5. When entering the scope (step *1.*), a class symbol is created and added to the current scope, i.e., the artifact scope

Java

```
1 public class C {
2   public void m() {
3     int v;
4   }
5   private String f;
6 }
```

Listing 5.5: Example class processed in Figure 5.6.

(step *1.1*). Next, the scope of the class is put on the stack, and hence, becomes the current scope (step *1.2*). Similarly, the method scope starts at line 2. When entering the scope (step *2.1*), a method symbol is added to the class scope (which is the current scope), and then, the method scope is put on the stack and becomes the current scope (step *2.2*). Thereafter, the local variable v (line 3, Listing 5.5) is handled (step *3.*). Since a variable does not span a scope, only its corresponding symbol is added to the current scope (i.e., the method scope). In line 4, the method scope ends (step *4.*), and thus, is removed from the stack. Consequently, the class scope becomes the current scope again.

Analogous to the local variable v, a symbol for the field f in line 5 is added to the current scope (step *5*). Since the class scope is the top element of the stack, f's symbol is added to C's spanned scope. Finally, line 6 closes the class scope, thus, the class scope is removed from the scope stack (step *6*). As initially (step *0.*), the artifact scope again is the only scope on the stack (but now contains the whole symbol table of class C).

Please note that the order of the steps is important. For example, the symbol representing field f must be added to the current scope C (step *5*) *before* C is removed from the stack (step *6.*). Otherwise, f would be added to the artifact scope.

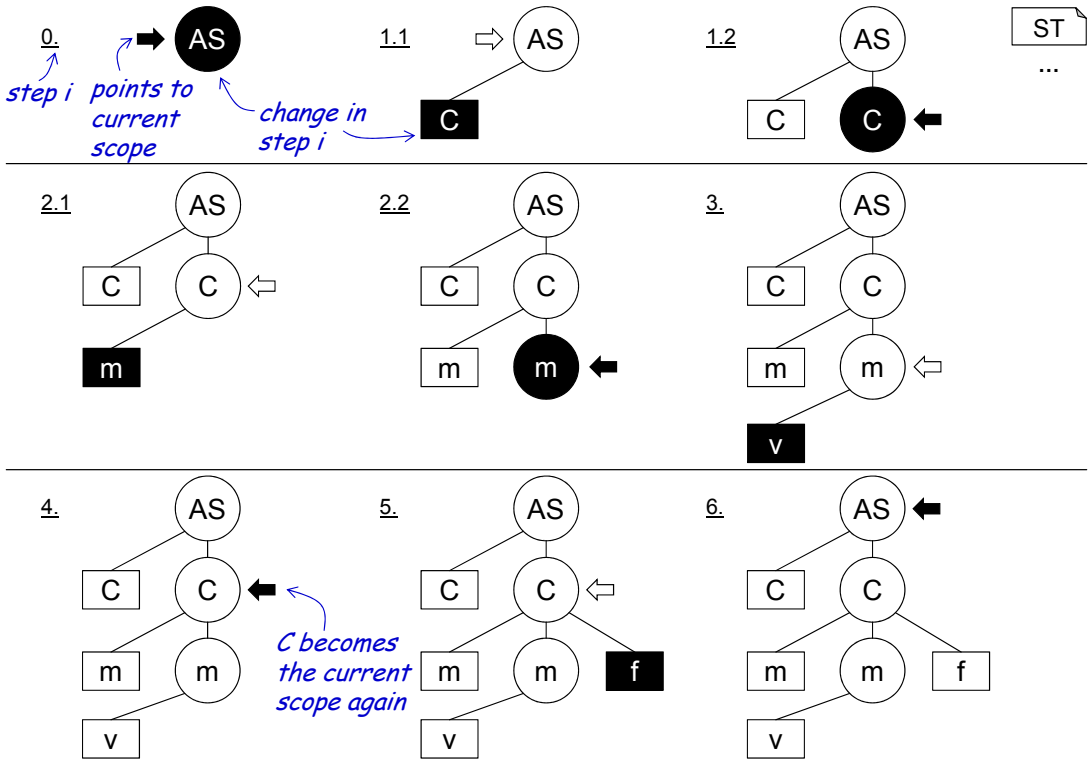


Figure 5.6: Example of a stack-based symbol table creation, which incrementally builds up a scope graph and the contained symbols of the class in Listing 5.5.

5.3.2 Method for Technical Realization of a Symbol Table Creator Using the Visitor Pattern

Technically, traversing the model means traversing its corresponding AST which is typically conducted by visitors [GHJV95] as already introduced in Section 2.2.4. As an example, for step *1.1* in Figure 5.6 the method `visit(ASTClassDeclaration)` is used, which is invoked every time a class scope is entered. To handle the exiting of a scope, the `endVisit(ASTClassDeclaration)` method can be employed. This method is invoked every time the processing of an AST node ends (steps *4.* and *6.* in Figure 5.6).

Figure 5.7 gives a method for deciding whether a `visit` and an `endVisit` method should be implemented for a specific AST node. That means, it helps to answer the following question: *Given the AST node of a model element, which visitor methods (i.e., `visit` and `endVisit`) should be implemented for it in the symbol table creator?* The implementation should follow the methods given in Section 5.2.

If an AST node represents a model element of case **(a)**, i.e., it is not related to any symbol table element, neither a `visit` method nor an `endVisit` method needs to be implemented. On the contrary, the cases **(b)**, **(c)**, and **(d)** each require at least a `visit` method based on the general approaches depicted in Figure 5.2, Figure 5.3, and Figure 5.4, respectively.

Furthermore, if the model element spans a scope (i.e., cases **(b)** and **(d)**), the scope is put on the stack (in order to become the current scope). Consequently, a corresponding `endVisit` method should be implemented that removes the scope from the stack when the processing of the AST node finished.

Section 7.9 demonstrates how the method depicted in Figure 5.7 is used for generating a language-specific symbol table creator.

5.4 Linking AST Nodes and Symbol Table Elements

Phase **P4**—that is, linking AST nodes and corresponding symbol table elements—is conducted in two phases, a language-specific phase and a language-unspecific phase:

P4.1 (language-specific) In the first phase, the link is set for all AST nodes that are *processed directly*, i.e., belong to one of the cases **(b)-(d)** described in the Subsections 5.2.2, 5.2.3, and 5.2.4. Technically, this means that a corresponding (non-empty) `visit` method exists (cf. Section 5.5). This phase is language-specific since the link between the AST node and the symbol table element has to be explicitly stated and requires language-specific knowledge, for example, that a `JavaTypeSymbol` is to be linked with an `ASTInterfaceDeclaration`. However, linking symbols and AST nodes via SMI can be conducted generically as described in Section 5.5.

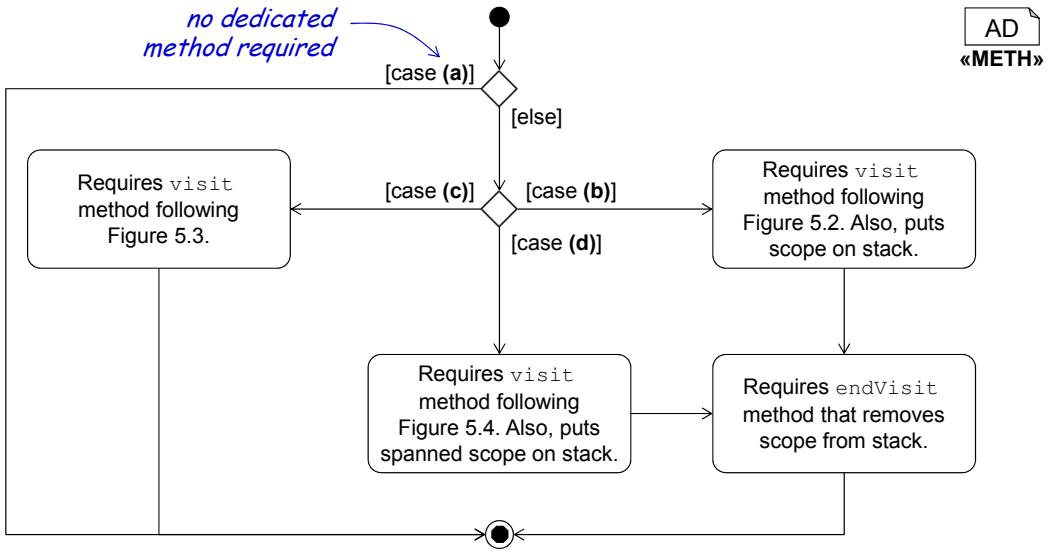


Figure 5.7: Method for determining which visitor methods in the symbol table creator are required for a given model element.

P4.2 (language-unspecific) Next, depending on the links set in phase **P4.1**, the enclosing scope for all remaining AST nodes can be set in a generic way since it does not rely on any language-specific information but solely on an AST node's context, i.e., its parent node(s) [Völ11]. Figure 5.8 shows the general process for determining the enclosing scope of a node n. The root AST node is skipped since it has no parent node. Typically, it is linked to an artifact scope, which is the root of a model's scope graph (cf. Section 3.5.5). As a result, its enclosing scope is the global scope. If n's parent node p spans a scope, n's enclosing scope is p's spanned scope. Else, if p does not span a scope, n's enclosing scope is the same as p's enclosing scope.

It is important that phase **P4.2** is conducted *after* phase **P4.1** since the links set in **P4.2** highly depend on links already set in **P4.1**.

In the following the two phases are illustrated by an example. Figure 5.9 shows a simplified object-diagram for the AST and symbol table of a Java interface declaration. After phase **P4.1**, `ASTInterfaceDeclaration` is linked to its corresponding `JavaTypeSymbol` and its spanned scope `JavaTypeScope`[2] (case **(d)**, cf. Section 5.2.4). Similarly, the `ASTFieldDeclaration` node is linked with the symbol `JavaFieldSymbol` (case **(c)**, cf. Section 5.2.3). Additionally, the spanned scope of the interface declaration is set as enclosing scope of `ASTFieldDeclaration`. Since `ASTPrimitiveModifier` neither is represented by a symbol nor does span a scope (case **(a)**, cf. Section 5.2.1), it is not linked to any symbol table element in this first phase.

[2]Pattern *(D) Same Symbol Class for Similar Model Elements* (cf. Section 4.1.2) is assumed.

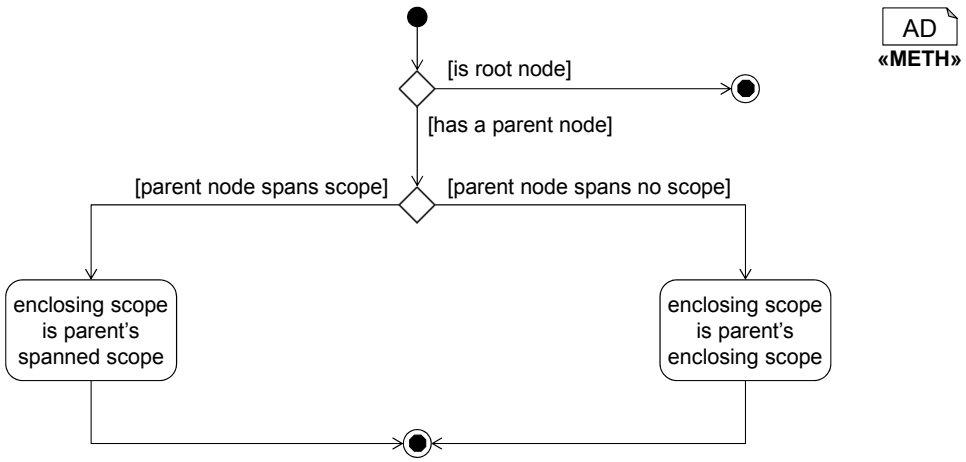


Figure 5.8: Method for linking AST and ST elements in phase **P4.2**.

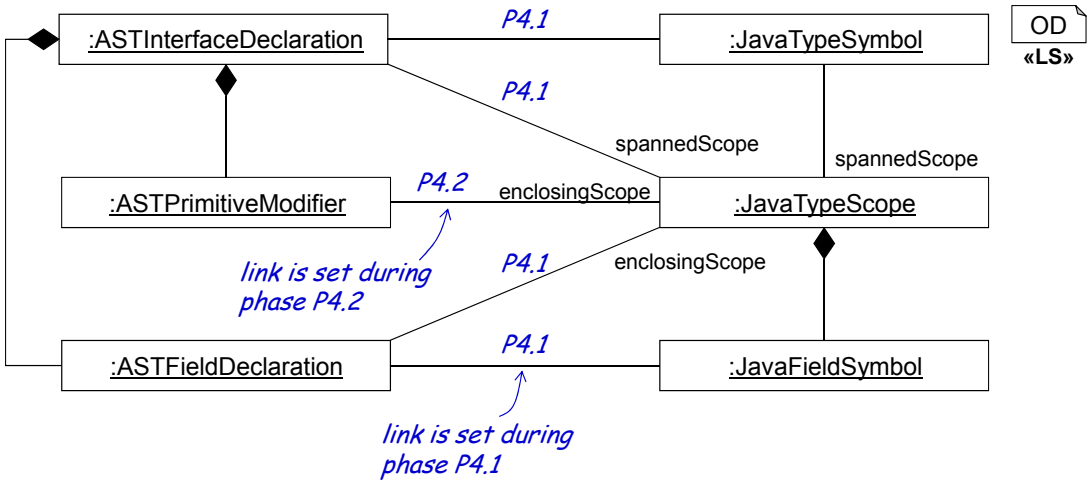


Figure 5.9: Exemplary linking of AST and ST elements during the phase **P4**.

In the second phase **P4.2**, the previously set links are utilized to determine the enclosing scope of the remaining `ASTPrimitiveModifier` node. Hence, the enclosing scope of `ASTPrimitiveModifier` is the spanned scope of its parent node `ASTInterfaceDeclaration` (cf. left case in Figure 5.8)

Please note that in contrast to phase **P4.2**, **P4.1** is conducted while processing the corresponding AST nodes (in the respective `visit` methods), as described in the previous section. Alternatively, the linking of AST nodes and symbol table elements (i.e., phase **P4**, Section 5.1) can be completely separated from the other symbol table creation steps, as follows:

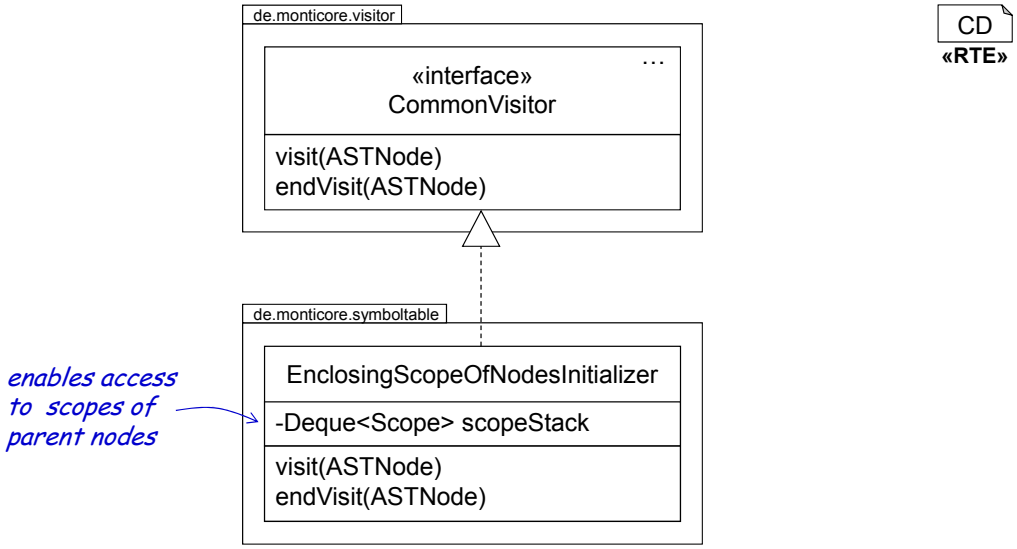


Figure 5.10: Class `EnclosingScopeOfNodesInitializer` for conducting phase **P4.2** based on method outlined in Figure 5.8.

1. Conducting symbol table creation phases **P1**, **P2**, and **P3** (cf. Section 5.1) for each AST node of the model, without setting any links between AST and ST.
2. Linking AST nodes of cases **(b)**, **(c)**, and **(d)** (cf. Section 5.2) with corresponding symbol table elements (i.e., phase **P4.1**).
3. Linking remaining AST nodes with corresponding enclosing scope (i.e., phase **P4.2**).

This approach separates concerns and allows to easily disable or enable the linking. At the same time, it introduces redundancy when handling AST nodes of the cases **(b)**-**(d)**. First, when the symbol table is created, the language engineer employs the AST nodes to create the corresponding symbol table elements (phases **P1**-**P3**). Second, in the linking phase **P4.1**, those AST nodes are linked to the previously created symbols and scopes. Technically, this means, that besides a visitor for the symbol table creation a second visitor for the linking is required which implements `visit` methods for the same AST nodes. In consequence, an additional traversal of the AST must be conducted.

5.4.1 Technical Realization of Language-Unspecific Linking

SMI provides the `EnclosingScopeOfNodesInitializer` class which ultimately performs the (language-unspecific) linking of the remaining AST nodes (i.e., case **(a)**) in phase **P4.2**. As depicted in Figure 5.10, `EnclosingScopeOfNodesInitializer` implements the `CommonVisitor` interface provided by MontiCore which, among others,

has a `visit` and an `endVisit` method. The `visit` method links all remaining AST nodes to their corresponding enclosing scopes, based on the method depicted in Figure 5.8. Additionally, `EnclosingScopeOfNodesInitializer` manages a stack which contains the current scope as its top element. The `endVisit` method solely removes the top element if it finished processing (cf. Section 5.3).

Please note that MontiCore's AST omits links to parent nodes. The stack, however, enables to retrieve information associated with the parent nodes, e.g., the enclosing scope.

5.5 Implementing a Language-Specific Symbol Table Creator

SMI provides the interface `SymbolTableCreator` in order to create language-specific symbol tables. Its default implementation `CommonSymbolTableCreator` conducts the language-unspecific tasks mentioned earlier in this section, i.e., creating the scope tree (using a stack), adding symbols to the enclosing scope, and linking AST and symbol table elements. Figure 5.11 highlights the corresponding methods (and their default behavior), which are:

getFirstCreatedScope() Returns the scope that has been created at first during the symbol table creation of the *current model* (or model element). Within a single language, this is usually the artifact scope (i.e., the root scope of a model). However, when applying language embedding, the root scope depends upon the embedded model element.

putOnStack(MutableScope) Puts a scope onto the stack. If the stack was empty, the added scope is the first created scope (see method `getFirstCreatedScope`). Else, the enclosing-sub relation between the newly added scope and the (previously) top scope is set (cf. *set enclosing-sub scope relation*, Figure 5.2). The parameter is of type `MutableScope` in order to enable to set the enclosing-sub relation (cf. Section 4.2.1). Additionally, scope-specific resolving filters are set in this method (see below).

currentScope() Returns the current scope, i.e., the top element of the scope stack. Since the symbol table is created incrementally (cf. Section 5.3), this method helps to access the scope that is currently processed.

currentSymbol() If the current scope is spanned by a symbol, this method returns the spanning symbol.

removeCurrentScope() Removes the current scope from the stack. The next scope on the stack becomes the new current scope.

addToScope(Symbol) Adds the symbol to the current scope (cf. *add symbol to enclosing scope*, Figure 5.3). If no current scope exists, a warning will be issued.

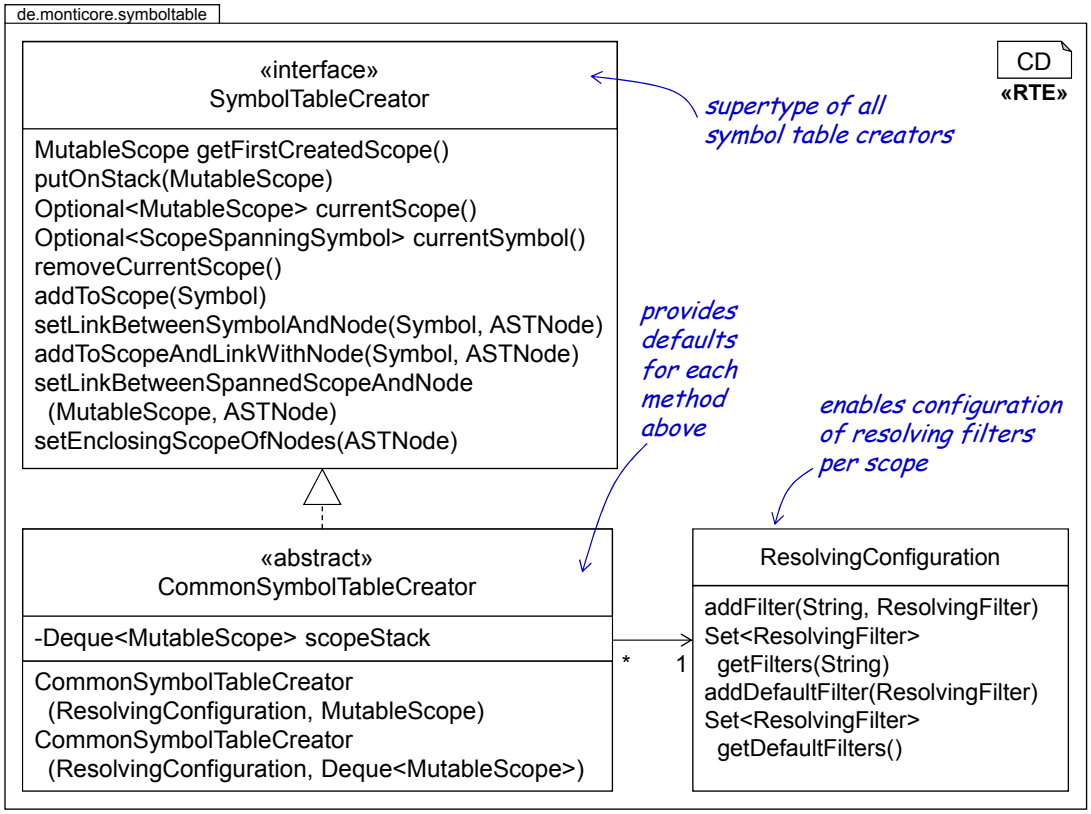


Figure 5.11: `SymbolTableCreator` and its default implementation `CommonSymbolTableCreator`, which employs `ResolvingConfiguration` to configure the resolving filters of scopes.

setLinkBetweenSymbolAndNode(Symbol, ASTNode) As part of **P4.1**, this method links together the symbol and the AST node (cf. *link symbol and node*, Figure 5.3). Additionally, the AST node's enclosing scope is set to be same as the symbol's enclosing scope.

addToScopeAndLinkWithNode(Symbol, ASTNode) This method groups the methods `putOnStack`, `addToScope`, and `setLinkBetweenSymbolAndNode`, in order to conduct the last two activities of Figure 5.3 and the last three activities of Figure 5.4 (if the symbol spans a scope), respectively. This ensures that the correct order is preserved while incrementally building up the symbol table and prevents inconsistencies resulting from wrong ordering (cf. Section 5.3.1).

setLinkBetweenSpannedScopeAndNode(MutableScope, ASTNode) Sets the relation between an AST node and its spanned scope (cf. *link scope and node*, Figure 5.2), as part of phase **P4.1**. For this, a mutable scope is required.

setEnclosingScopeOfNodes(ASTNode) Conducts phase **P4.2** for all AST nodes by executing `EnclosingScopeOfNodesInitializer` starting from the given root node (cf. Section 5.4.1).

CommonSymbolTableCreator(ResolvingConfiguration, MutableScope) The first parameter of this constructor initializes the symbol table creator with a configuration for resolving filters (see below). The second parameter sets the enclosing scope of the model (element) whose symbol table is to be created. Since the root scope within a model typically is an artifact scope (cf. Section 3.5.5), the passed enclosing scope usually is the global scope (cf. Section 3.5.6). In general, this constructor is invoked when creating symbol tables within a single language.

CommonSymbolTableCreator(ResolvingConfiguration, Deque<MutableScope>) Same as for the previous constructor, the first parameter initializes the symbol table creator with a configuration for resolving filters. The second parameter contains a stack of (enclosing) scopes. In particular, this stack is required for language embedding to share the same scope stack among symbol tables of different languages (cf. Section 8.2).

Please note that `CommonSymbolTableCreator` manages a stack of *mutable scopes* since, among others, the enclosing-sub relation needs to be set (cf. Section 4.2.1).

In addition to the methods above, every symbol table creator must provide (at least) a method `createFromAST` with a language-specific AST node as the only parameter. This will be demonstrated in Section 5.5.

Resolving Configuration

The class `ResolvingConfiguration` depicted in Figure 5.11 serves as configuration for the resolving filters introduced in Section 5.2. It allows to customize resolving filters per scope, which determine the resolvable symbols starting from a specific scope. The customization is managed via the method `addFilter(String, ResolvingFilter)`, which configures the resolving filter for a scope with the given name. `getFilters(String)` returns all resolving filters previously set for a scope. If no specific resolving filters are specified for a scope, it obtains the resolving filters of its enclosing scope (if exists), or else the default filters set and retrieved via `addDefaultFilter(ResolvingFilter)` and `Set<ResolvingFilter> getDefaultFilters()`, respectively.

The next section demonstrates how the cases presented in Section 5.2 can be implemented with the methods provided by `CommonSymbolTableCreator` using the example of a simplified Java AST structure.

Example of a Language-Specific Symbol Table Creator

Figure 5.12 depicts an excerpt from the (simplified) AST structure of Java. Its root node is `ASTCompilationUnit` which contains package information and import statements. Furthermore, it consists of interface declarations which in turn can define inner interfaces and field declarations. Both interfaces and fields can have modifiers. Please note that each AST node subtypes the (generated) interface `ASTJavaNode` (cf. Section 2.2.4) which is important for the `createFromAST` method described below.

Listing 5.13 shows the structure of the symbol table creator for the Java language, namely `JavaSymbolTableCreator`. As it can be seen, `JavaSymbolTableCreator` extends `CommonSymbolTableCreator` (line 2) in order to reuse the default implementations. Also, it implements the (generated) visitor interface `JavaVisitor` (line 3), which enables to handle any AST node of the language as needed (cf. Section 2.2.4).

Besides the constructor of its superclass (lines 5–9), `JavaSymbolTableCreator` provides the `createFromAST` method (lines 11–14), which creates the symbol table starting *from any AST node of the Java language* (line 12) by using the parameter type `ASTJavaNode`. Finally, `createFromAST` returns the first created scope via `getFirstCreatedScope` (line 13). This enables a functional-like usage of the symbol table creator: given an AST the corresponding symbol table can be constructed via `stCreator.createFromAST(ast)`.

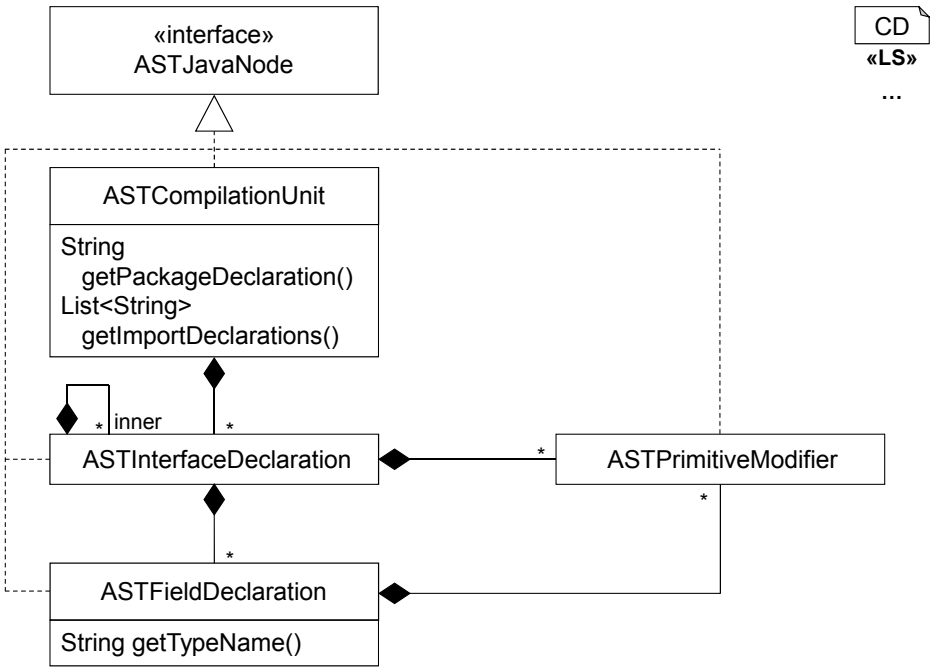


Figure 5.12: Simplified AST structure for Java.

```java
public class JavaSymbolTableCreator
             extends CommonSymbolTableCreator
             implements JavaVisitor {

  public JavaSymbolTableCreator(
      ResolvingConfiguration resConfig,
      MutableScope enclosingScope) {
    super(resConfig, enclosingScope);
  }

  public Scope createFromAST(ASTJavaNode rootNode) {
    rootNode.accept(this);
    return getFirstCreatedScope();
  }

  //...
}
```

Java «LS»

Listing 5.13: Implementation of `createFromAST` method for `JavaSymbolTableCreator`.

```java
@Override
public void visit(ASTCompilationUnit ast) {
  //--- activities "create scope" and "initialize scope" ---//
  String packageName = getPackage(ast.getPackageDeclaration())
  List<ImportStatement> importStatements =
    getImports(ast.getImportDeclarations());
  ArtifactScope artifactScope =
    new ArtifactScope(packageName, importStatements);

  //--- activity "set enclosing-sub scope relation" ---//
  putOnStack(artifactScope);

  //--- activity "link scope and node" ---//
  setLinkBetweenSpannedScopeAndNode(artifactScope, ast);
}
```

Java «LS»

Listing 5.14: Implementation of `visit` method for `ASTCompilationUnit`.

Implementation for `ASTCompilationUnit` (Case (b) on Page 103)

In general, an artifact scope is created when processing the root node of a model, which in the current example is of type `ASTCompilationUnit`. Listing 5.14 presents the implementation of the corresponding `visit` method. Following from Figure 5.2, the process starts with creating (cf. *create scope*, Figure 5.2) and initializing (cf. *initialize scope*, Figure 5.2) an `ArtifactScope` (lines 4–8, Listing 5.14). The package information

as well as the import statements are important for the inter-model resolution processes and will be elaborated in Section 6.3 and Section 6.4. Next, the newly created artifact scope is put on the stack (line 11) which (optionally) sets the enclosing-sub relation (cf. *set enclosing-sub scope relation*, Figure 5.2). Finally, the artifact scope and its corresponding AST node are linked together (line 14, cf. *link scope and node*, Figure 5.2).

The `endVisit` method of `ASTCompilationUnit` is invoked when the processing of all other AST nodes finished. Hence, besides removing the artifact scope from the stack (line 3, Listing 5.15), the enclosing scope of all remaining AST nodes is set via `setEnclosingScopeOfNodes`, that means, phase **P4.2** is conducted (cf. Section 5.4.1).

```
1 @Override
2 public void endVisit(ASTCompilationUnit ast) {
3   removeCurrentScope();
4   setEnclosingScopeOfNodes(ast);
5 }
```

Java «LS»

Listing 5.15: Implementation of `endVisit` method for `ASTCompilationUnit`.

Implementation for `ASTFieldDeclaration` (Case (c) on Page 104)

A Java field is represented by a symbol (cf. case **(c)**, Section 5.2) but does not span a scope, and hence, is processed as described in Section 5.2.3. Listing 5.16 presents the implementation of the corresponding `visit` method. First, a `JavaFieldSymbol` is instantiated (lines 6–7), following the *create symbol* activity in Figure 5.3. Since a `JavaFieldSymbol` always has a type, it is also initialized with a `JavaTypeSymbolReference` (lines 4–5). The scope that defines the type reference (here, an interface scope) is obtained via `currentScope()`. Next, further initialization is conducted (cf. *initialize symbol*, Figure 5.3), e.g., setting the modifiers (line 11). Thereafter, the symbol is added to its enclosing scope (line 14, cf. *add symbol to enclosing scope*, Figure 5.3) and linked to its corresponding AST node (line 17, cf. *link symbol and node*, Figure 5.3).

As already mentioned, references (e.g., the field's type reference) should not be resolved during the symbol table creation (see phase **P5**), but after the symbol table creator finished, e.g., in the `endVisit` of the root AST node `ASTCompilationUnit`.

Implementation for `ASTInterfaceDeclaration` (Case (d) on Page 104)

Listing 5.17 shows the processing of `ASTInterfaceDeclaration`—which is a model element of case **(d)** (cf. Section 5.2)—according to the procedure depicted in Figure 5.4.

Since `JavaTypeSymbol` subclasses `CommonScopeSpanningSymbol`, the first three activities of Figure 5.4 (namely, *create spanned scope*, *create symbol*, and *link symbol and scope*) are already conducted implicitly when instantiating it (line 5).

```java
@Override
public void visit(ASTFieldDeclaration ast) {
  //--- activity "create symbol" ---//
  JavaTypeSymbolReference typeRef = new JavaTypeSymbolReference
    (ast.getTypeName(), currentScope().get());
  JavaFieldSymbol fieldSymbol =
    new JavaFieldSymbol(ast.getName(), typeRef);

  //--- activity "initialize symbol" ---//
  // sets modifiers such as public, private, final, static, ...
  setModifiers(fieldSymbol, ast.getModifiers());

  //--- activity "add symbol to enclosing scope"  ---//
  addToScope(fieldSymbol);

  //--- activity "link symbol and node"
  setLinkBetweenSymbolAndNode(fieldSymbol, ast);
}
```

Java «LS»

Listing 5.16: Implementation of `visit` method for `ASTFieldDeclaration`.

Next—given that pattern *(D) Same Symbol Class for Similar Model Elements* (cf. Section 4.1.2) is applied[3]—the symbol is set to be an interface (line 8). Also, it is defined as abstract (line 11) following the semantics of an interface in Java (cf. essential model, Def. 3.15) which is always abstract [GJS+14] (although not explicitly stated in the model).

Please note that `visit(ASTInterfaceDeclaration)` in Listing 5.17 does not handle an interface's fields. Instead, these are handled by their own `visit` method (i.e., Listing 5.16), following the top-down approach presented in Section 5.3.

Finally, `addToScopeAndLinkWithNode` conducts the last three activities of Figure 5.4, i.e., adding `typeSymbol` to its enclosing scope, setting the enclosing-sub relation of `typeSymbol`'s spanned scope and the enclosing scope, and linking the `typeSymbol` and its spanned scope with the `ast` node (line 22).

By using the top-down symbol table creation approach (cf. Section 5.3), we do not have to distinguish whether `typeSymbol` represents a top-level interface or an inner interface when adding it to its enclosing scope. Since the enclosing scope (whether artifact scope or a type scope) is already handled, it is the current scope on the stack. Hence, `typeSymbol` is added to the correct scope (cf. Figure 5.6). Finally, the `endVisit` method for `ASTInterfaceDeclaration` (not shown here) removes `typeSymbol` from the stack via `removeCurrentScope()`.

[3] In case of pattern *(E) Different Symbol Classes for Similar Model Elements* (cf. Section 4.1.2), `JavaInterfaceSymbol` would be instantiated instead of `JavaTypeSymbol`. Consequently, further initialization, such as in line 8 and line 11 (cf. Listing 5.17), would not be required.

```java
@Override
public void visit(ASTInterfaceDeclaration ast) {
  //--- activities "create spanned scope", "create symbol", and
  //    "link symbol and scope" ---//
  JavaTypeSymbol typeSymbol = new JavaTypeSymbol(ast.getName());

  //--- activity "initialize symbol and scope" ---//
  typeSymbol.setInterface(true);

  // an interface is always abstract
  typeSymbol.setAbstract(true);

  // sets further modifiers such as public or package-local
  setModifiers(typeSymbol, ast.getModifiers());

  // further initialization, such as setting the super types, etc.
  // ...

  //--- activities "add symbol to enclosing scope",
  //    "set enclosing-sub scope relation", and
  //    "link node with symbol and scope" ---//
  addToScopeAndLinkWithNode(typeSymbol, ast);
}
```

Java «LS»

Listing 5.17: Implementation of `visit` method for `ASTInterfaceDeclaration`.

The initialization of the interface's modifiers (line 14, Listing 5.17) is an example of case **(a)**. That means, `ASTPrimitiveModifier` is not related to any ST element but solely used in order to initialize `typeSymbol`. According to Figure 5.7, a dedicated `visit(ASTPrimitiveModifier)` method is not required.

Exchangeable Instantiation of Symbols

Instantiating the symbols and references directly via the `new` operator, as done in the example above, hampers, above all, language inheritance (cf. Section 8.4). It is, for example, not possible to substitute the instantiation of `JavaTypeSymbol` (line 5, Listing 5.17) with an instantiation of a subtype. However, this is essential since the extending language can add further information to a model element. MontiJava, for example, extends the Java language and adds new modifiers such as `singleton`, in order to specify singleton classes [Mul15]. Since the other properties of a class remain unchanged, it should be possible to completely reuse the corresponding symbol table creator of Java and extend it with MontiJava specific information.

To enable this, any of the patterns Abstract Factory, Factory Method, and Builder as presented by Gamma et al. [GHJV95] are suited and should be applied for the symbol table creator of a language. Section 7.9 demonstrates the generation of a symbol table creator using the Factory Method pattern.

5.6 Comparison to Symbol Table Creation in Previous MontiCore Versions

Völkel [Völ11] employs dedicated symbol table kinds for the different access modifiers of a symbol (cf. Section 3.10). Consequently, the symbol table engineer must be aware of those different kinds when processing an AST node.

Listing 5.18 illustrates an excerpt from the `visit(ASTInterfaceDeclaration)` method implemented with previous MontiCore versions based on the symbol table approach presented in [Völ11]. First, the associated namespace of the node (cf. Section 4.6) is initialized with (empty) symbol tables for each kind via the `init` method (line 6, Listing 5.18). The method is language-specific, and thus, must be implemented for each language. The instantiation and initialization of the entry (lines 8–13) is similar to Listing 5.17. Next, the interface entry is added to each exported symbol table of its enclosing namespace depending on the interface's modifier (lines 16–22). For example, if it is protected, the entry is added to each exported symbol table of the kinds `public` and `protected`. Finally, the symbol is added to the stack, which allows for retrieving the current symbol, same as the `currentSymbol` method of our approach.

Please note that the lines 16–25 perform a similar task as `addToScope` (cf. Section 5.5). However, since in SMI access modifiers are top-level concepts, the symbol table engineer is liberated from handling different symbol table kinds during the symbol table creation process. Instead, modifiers are *automatically* considered during the resolution process (cf. Chapter 6). Moreover, the inclusion relation of access modifiers (cf. Def. 3.14) enables to easily filter not included symbols during symbol resolution. Hence, a method, such as `getAllExportedTables` in line 17 of Listing 5.18, is not required. This simplifies the symbol table creation and prevents that wrong symbol table kinds are used.

Furthermore, the `storage` employed in Listing 5.18 also serves for the qualification of references. After the symbol table creation, another workflow qualifies all references contained in `storage`. This approach yields two disadvantages. First, the symbol table engineer must be aware of adding references not only to the respective entries but also to the storage, which can be a typical source of errors. Second, emerging from the workflow-based architecture of MontiCore before version 4, it is not possible (at least not with little effort) to create the symbol table in a functional way, as with the `createFromAST` method described above. SMI eliminates both problems by (lazily) qualifying references during the resolution process. This is discussed in Section 6.3.

Java «LS»

```
1 public void visit(ASTInterfaceDeclaration node) {
2   // ...
3
4   // Among others, initializes the associated namespace
5   // with the different symbol table kinds.
6   init(node);
7
8   JavaTypeEntry te = createTypeFromNode(node);
9   te.setEntryState(STEntryState.FULL, te);
10  setTypeModifiers(modifier, te);
11
12  te.setInterface(true);
13  te.setAbstract(true);
14
15  // All exported symbol tables of the enclosing namespace.
16  List<SymbolTable> tables =
17    getAllExportedTables(modifier, node.get_Parent());
18
19  // Add entry to each exported symbol table.
20  for (SymbolTable table : tables) {
21    table.addEntry(te);
22  }
23
24  // Similar to addToScope(Symbol)
25  storage.getEntryStack().push(te);
26 }
```

Listing 5.18: Implementation of `visit` method for `ASTInterfaceDeclaration` based on approach in [Völ11].

Chapter 6

Symbol Resolution in SMI

Section 3.2 already describes the importance of names in software languages (cf. [GM10]). They are, among others, used to refer to elements defined elsewhere (cf. Section 3.4). Moreover, a model's interface (determined by its modeling language's interface, cf. Section 3.8) consists of names and associated information [Rum13] and can be employed by other models. By this means, names enable *name-based composition* of models [Rum13, HR13].

The underlying process is called name resolution[1] which "associates each reference to its intended declaration(s), according to the semantics of the language" [NTVW15]. A language's semantics regarding name resolution specifies, for instance, whether forward referencing is allowed. This in turn determines whether the exact occurrences (i.e., source position) of a reference in its enclosing scope is relevant. Furthermore, the scope types (e.g., shadowing and visibility scopes, cf. Def. 3.8) a language provides, also affect the name resolution process. Moreover, the binding time is essential. While *static binding* takes place before the program is executed, e.g., during compile time, *dynamic binding* occurs during run time [Seb08, GM10]. This thesis focuses on static name binding (cf. Chapter 3).

The goal of this chapter is to present exemplary use cases and general processes for name resolution based on language features—determined by the language's semantics—introduced in Chapter 3, such as symbol shadowing and access control mechanisms. Moreover, this chapter elucidates how SMI standardly realizes those use cases based on the technical basis formed by the previous chapters: Chapter 4 presents technical classes for relevant concepts (such as symbols, symbol references, and scope types) and Chapter 5 exploits them to build up the scope graph (on which then the name resolution is conducted). Furthermore, this chapter demonstrates how a language engineer can adapt the name resolution in order to meet language-specific requirements. Therefore, this chapter introduces reference implementations that are (at least conceptually) reusable when developing own resolution strategies.

Same as [Par10, VS10, Völ11, Bet13], the current thesis applies an imperative approach for defining the strategy of the resolution process. Some approaches use declarative techniques. Gabriël et al. [KKWV13], for example, developed the meta-language *Name*

[1] We use the terms "name resolution", "name binding", and "symbol resolution" interchangeably.

Binding Language (NaBL), which provides first-level name binding concepts for, among others, name definition and usages, and scoping rules.

The theory our name resolution is based on has a lot in common with the work conducted by Neron et al. [NTVW15]. In particular, our resolution approach is determined by the underlying scope graph. A reference can only be resolved to its definition if there exists a path from the reference to the definition on the scope graph. Please refer to [NTVW15] (and its extensions [VANT+15a, VANT+15b]) for a comprehensive introduction of the name resolution theory. Further discussions can be found in Chapter 3. A difference of our approach to [NTVW15] is that the current thesis mainly focuses on file-based languages where artifact scopes play an essential role. Therefore, SMI introduces the explicit concept of an artifact scope (cf. Section 3.5.5). In contrast, Neron et al. do not treat artifact scopes as top-level concepts (but still provide ways to apply them via a combination of other concepts). Furthermore, Neron et al. do not explicitly distinguish between symbol kinds, which is a fundamental concept of the current thesis.

6.1 Overview and Primary Requirements

This section gives an overview of the current thesis' symbol resolution process by the example of class `java.lang.System`[2] of the JDK. For reasons of clarity, the excerpt shown in Listing 3.1 is repeated in Listing 6.1.

Java «MODEL»

```
1 package java.lang;
2 // ...
3 public final class System {
4   // ...
5   private static Properties props;
6   // ...
7   public static String getProperty(String key) {
8     checkKey(key);
9     SecurityManager sm = getSecurityManager();
10    if (sm != null) {
11      sm.checkPropertyAccess(key);
12    }
13    return props.getProperty(key);
14  }
15  // ...
16 }
```

Listing 6.1: Excerpt from the `java.lang.System` class (as in Listing 3.1).

[2] see `https://docs.oracle.com/javase/8/docs/api/java/lang/System.html`

Considering the `System` class, it can be distinguished between referenced model elements (or symbols) that are defined within `System` and those that are defined in another artifact (or model). `props`, for example, is used within the method scope (line 13). To determine the local variable or field the name `props` refers to, the search begins in the method scope (i.e., the innermost scope). Since `getProperty` does not define a local variable named `props`, the search continues in the enclosing class scope where a matching field is found. Consequently, the name `props` in `getProperty` refers to the same-named field defined in `System`. In contrast, `SecurityManager` used as type for the local variable `sm` (line 9) is not defined within `System` but in its own artifact. Hence, it can neither be found in the method scope nor in the class scope or the enclosing artifact scope. Thus, the search continues in the global scope (e.g., the class path). Since `SecurityManager` is an unqualified name (cf. Section 3.2), its qualified name (i.e., `java.lang.SecurityManager`) has to be determined first. In the current example, this can be conducted with the help of the package information (line 1). Having the qualified name, the definition of the `SecurityManager` class can be easily located.

To summarize, the name resolution starts in the innermost scope and continues with the enclosing scope until a corresponding symbol definition is found *within* the artifact. If the definition is not found in the artifact, the search continues in the global scope using additional information, such as the package name and the import statements. This approach is similar to the one Gabbrielli et al. call "static scope rule" or "rule of nearest scope" [GM10] (also cf. [ALSU06, Seb08, Völ11, KKWV13, NTVW15]) and very common in many languages (e.g., Java, C#, and MontiArc).

In the current thesis, the search within the model is called *intra-model resolution*, outside the model *inter-model resolution*, both of which are further divided as highlighted in Figure 6.2. The search from the `getProperty` (*gP*) method to the artifact scope is called *bottom-up intra-model resolution* since it takes place *within* the model and traverses the scope graph *bottom-up*. Next, the search from the artifact scope of `System` (*Sy*) to the global scope is the *bottom-up inter-model resolution* since it starts the search *outside* the model. After that, the *top-down inter-model resolution* continues with the artifact scope of `SecurityManager` (*Se*). Lastly, the resolution continues within the artifact scope of `SecurityManager` which is called *top-down intra-model resolution*. Each of these four phases exploits different information and techniques in order to conduct the search, as elaborated throughout this chapter.

The following list highlights some major requirements concerning the name resolution. More detailed features are described by means of examples in the respective sections.

RRQ1 (Reasonable Defaults) SMI's resolution mechanism should provide reasonable defaults for the language features introduced throughout this chapter.

RRQ2 (Efficiency of Customization) A language engineer should be able to conduct customizations of the provided default behavior (cf. *RRQ1*) with *little effort* in order to meet language-specific requirements.

RRQ3 (Same Behavior of Symbols) It is essential that all symbols that are defined in the same scope also "behave uniformly with respect to name resolution" [NTVW15] as already discussed in Section 3.5 and throughout Chapter 4. This, among others, means, that given two scopes S_1 and S_2, either all symbols of S_2 are visible within S_1 or none (and vise versa). The resolution mechanism must ensure this. Exceptions can occur due to access modifiers and symbol shadowing.

RRQ4 (Encapsulating Model Loading in Name Resolution Process) The name resolution is conducted on the logical scope graph which emerges from physical artifacts. The logical class name `java.lang.System`, for example, corresponds to the physical (relative) path `java/lang/System.java`. A language user has to be liberated from the physical aspects and only focus on the logical ones. Hence, model loading should be a (hidden) part of the resolution process.

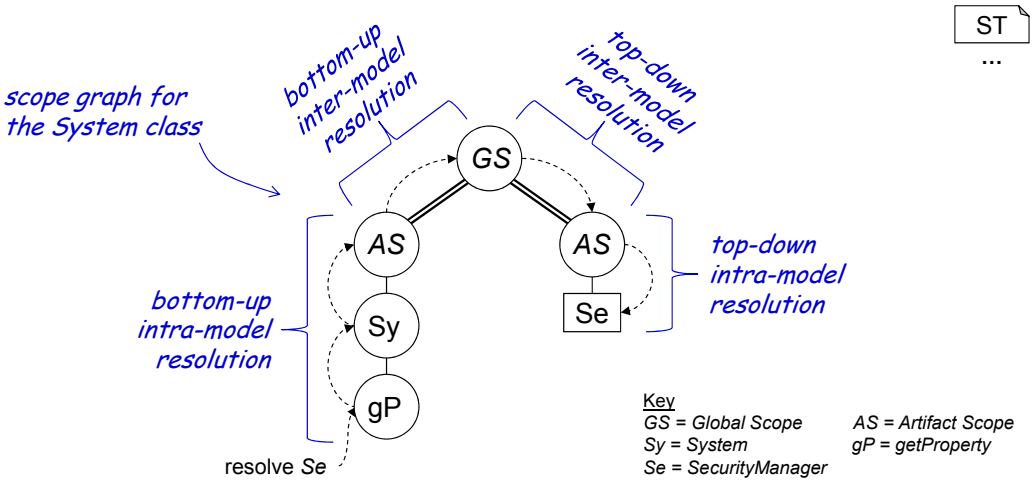


Figure 6.2: Overview of the four phases of the resolution process. In particular, the resolution (i) is conducted within a model (intra-model) or between models (inter-model) and (ii) traverses the scope graph bottom-up or top-down.

Chapter Outline

The remainder of this chapter is structured as follows. Sections 6.2, 6.3, 6.4, and 6.5 introduce for each phase of the name resolution process examples of some common language features as well as a general procedure. Next, Section 6.6 illustrates the resolution process in explicitly imported scopes and the difficulties they entail. Section 6.7 describes how the resolution can be further restricted via additional information, such as access modifiers. Subsequently, Section 6.8 presents the technical realization of the

features and concepts introduced in the previous sections. In Section 6.9 the model loading process and its integration in the resolution process are elucidated. Section 6.10 briefly demonstrates the configuration possibilities and usage of the resolution. Section 6.11 discusses further related approaches. Finally, Section 6.12 gives some naming conventions for language-specific implementations of the introduced technical classes.

6.2 Bottom-Up Intra-Model Resolution

In block structured languages (cf. Chapter 3) symbols are resolved starting from the innermost block to the outermost until a matching definition is found (e.g., [ALSU06, Seb08, GM10]). Furthermore, in many languages symbols may be shadowed in subscopes (cf. Section 3.5). For this, the subscope must be a shadowing scope. Figures 6.3, 6.4, and 6.5 illustrate some simple cases in which symbol shadowing can occur using the scope graph notation introduced in Section 3.9. In order to simplify understanding, some figures also depict exemplary Java code from which that scope graph could emerge.

In Figure 6.3 scope C and its subscope m both define a symbol named f. Furthermore, a reference to a symbol named f exists in m. Since the resolving process starts in the innermost scope, the first matching symbol is f defined in m. The search ends here since any symbol in m shadows same-named symbols (with the same kind) defined in its enclosing scope C.

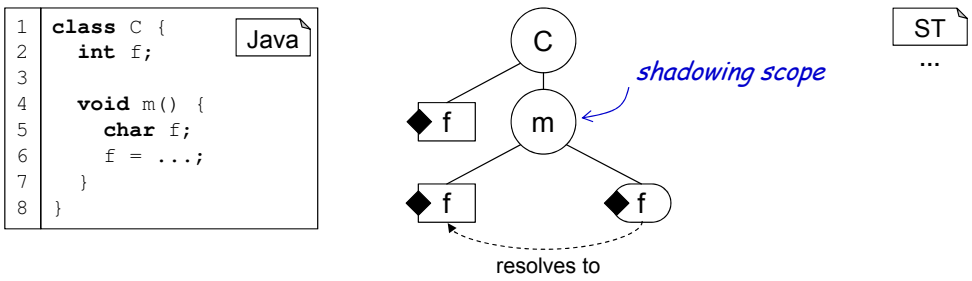


Figure 6.3: Exemplary bottom-up intra-model resolution in a shadowing scope.

The scope structure presented in Figure 6.4 is same as in Figure 6.3 with one crucial difference: the subscope is a visibility scope. Again, the search begins in the innermost scope where the symbol f matches. In contrast to the previous case, the resolution continues in m since the *if* scope is a visibility scope, and hence, its contained symbols do not shadow any symbols of m. As a consequence, the resolution result is ambiguous since both symbols match: f in the *if* scope and f in the m scope. In Java and C# this case results in a compilation error.

Lastly, Figure 6.5 shows a combination of the previous two cases. Again, f is defined in a visibility scope. Furthermore, a same-named symbol is defined in scope C. The shadowing scope m is in-between the C scope and the *if* scope. Same as before, f is

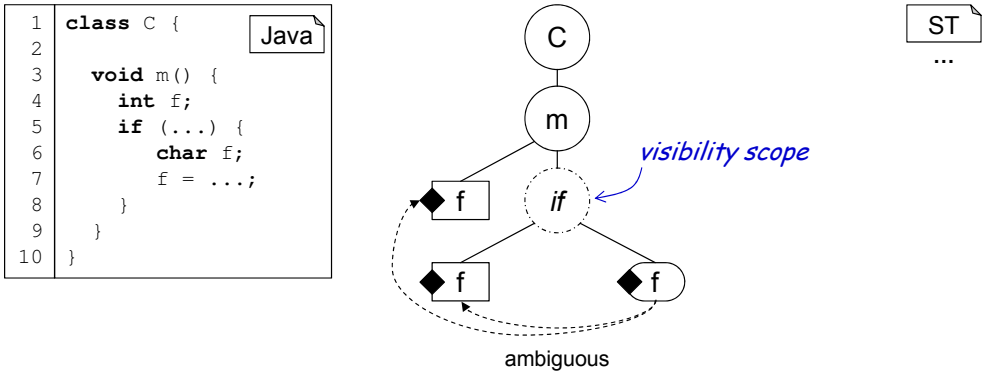


Figure 6.4: Exemplary bottom-up intra-model resolution in a visibility scope.

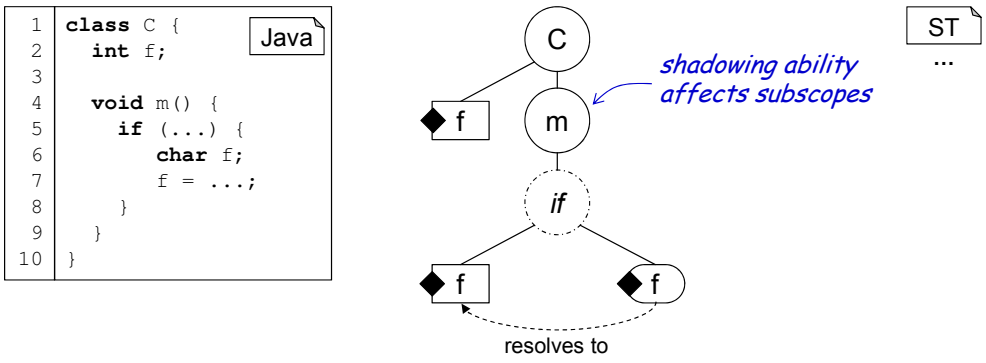


Figure 6.5: Exemplary bottom-up intra-model resolution in a visibility scope affected by the shadowing ability of its enclosing scope.

used in the *if* scope, hence, the search starts there and the first matching symbol is the *f* symbol defined locally. Since the *if* scope is a visibility scope, the search continues in *m*. Although no symbol is directly found in *m*, the search ends as *f* is already found in a subscope of *m*. This means, the shadowing ability of *m* also impacts results of its subscopes. Hence, the symbol *f* defined in the *if* scope shadows the same-kindred symbol *f* defined in the *C* scope.

The three examples of symbol shadowing presented in Figures 6.3, 6.4, and 6.5 can be combined to more complex examples, like a deeper scope hierarchy and more symbols with different kinds. However, the general resolution process remains the same, as described next.

General Process

As it can be seen in Figure 6.6, the bottom-up intra-model resolution process proceeds within a model (or artifact) and ends if the artifact scope is reached (or a matching symbol is found). The resolution starts in the innermost scope with the specified criteria which, among others, include name and kind of the searched symbol. Only symbols directly contained in that scope are considered (cf. *resolve locally*, Section 4.2.2). If (at least) one symbol matches the criteria and the current scope also is a shadowing scope or if no enclosing scope exists, the resolution stops with the found symbol(s) being the result (cf. Figure 6.3). If more than one symbol is found, the ambiguity can be handled (e.g., by throwing an exception, cf. Section 6.8.1).

If the current scope is a visibility scope or does not contain a matching symbol, the search will continue with the enclosing scope (if one exists), even if a symbol is found (cf. Figure 6.4). Consequently, the process illustrated in the activity diagram in Figure 6.6 is repeated upwards the scope hierarchy. This enables the case depicted in Figure 6.5, i.e., that the shadowing ability of a scope also affects its subscopes. Symbols found in the current scope and those found in its enclosing scope then are unified. Again, in case several symbols are found, ambiguity handling is required.

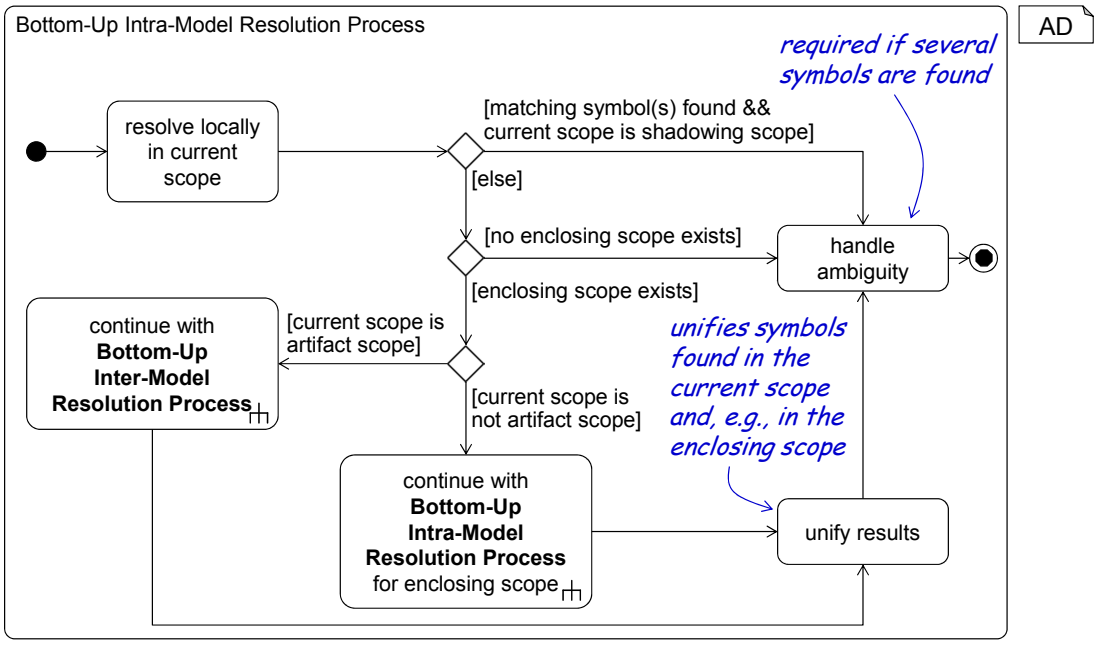


Figure 6.6: General process of the bottom-up intra-model resolution. If no symbol is found, the bottom-up inter-model resolution will continue (cf. Figure 6.8).

Please note that continuing with the enclosing scope enables automatic symbol importing, thus, symbols become visible in the subscopes (cf. open scope [GJR79, CL83]), as described in Section 3.10. If the artifact scope (cf. Section 3.5.5) is currently processed, the inter-model resolution phase will start (cf. Section 6.3). Finally, results of the inter-model resolution phase are unified with symbols found in the current intra-model resolution phase. If required, ambiguity is handled.

The bottom-up intra-model resolution process is similar in many frameworks. For example, [VS10, Par10, Völ11, Bet13] traverse the (scope or AST) hierarchy bottom-up until a matching element is found. Except for [Völ11], elements of inner nodes standardly shadow same-named elements of enclosing nodes. In [Völ11], this must be explicitly specified in a dedicated workflow which is previously conducted (cf. Section 5.1). In Spoofax [KV10], the shadowing ability is explicitly specified with a NaBL model [KKWV13].

6.3 Bottom-Up Inter-Model Resolution

If the specified symbol cannot be found within the model (or artifact), the inter-model resolution begins. For this, the artifact scope (cf. Def. 3.10) as well as the global scope (cf. Def. 3.11) play a special role. The former is the top scope within a model, and hence, determines how the resolution continues outside the model. The latter enables searching in other artifacts.

In contrast to the intra-model resolution, the symbol's unqualified name is not sufficient when searching outside the model, instead the fully qualified name is required. Many languages allow for specifying a symbol's unqualified name in combination with an import statement (cf. Section 3.5.5). From these, the fully qualified name is determined first, which is called *name qualification*.

In the following, the beginning of the qualification process is introduced by the example of Figure 6.7. The name qualification ends when a matching symbol is found which, however, can be in a different resolution phase. The C class refers to D via its simple name. There exist, among others, three possible fully qualified names for D.

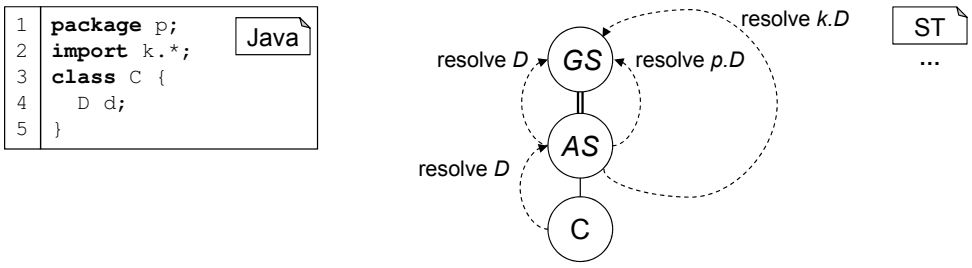


Figure 6.7: Exemplary bottom-up inter-model resolution including name qualification.

First, if D is in the same package as the referencing class C, its fully qualified name is $p.D$. Second, if it is defined in the (imported) package $k.*$, $k.D$ is the fully qualified name. Lastly, D might be a global type (such as `int` and `boolean`) or a type defined in an unnamed package (similar to the default package in Java [GJS+14]). In these cases the fully qualified name is the same as the simple name (i.e., D). Each of these different possible names need to be checked. Consequently, the resolution multiplies when resolving outside the artifact. The global scope conducts a local search (cf. Section 4.2.2) for each of these fully qualified names and—if no matching symbol is found—continues with a top-down inter-model resolution (cf. Section 6.4).

General Process

As depicted in Figure 6.8, the artifact scope starts the bottom-up inter-model resolution process which includes name qualification in case the searched name is unqualified. The artifact scope determines all potential names of a searched symbol based on the package and import information. For a simple name S potential names are:

- the simple name S itself (e.g., in case of global symbols),
- imported qualified names (via import statements) that end with the simple name S, e.g., $q.S$,
- $k.S$ for each star import $k.*$, and
- the artifact scope's package name p as the symbol's qualifier, i.e., $p.S$.

The global scope first tries to find global symbols (such as *int* and *boolean* in Java) by locally resolving each of the calculated potential names. If a corresponding symbol is found, it will be unified with previously found symbols (if any). More precisely, each symbol matching one of the potential names, is part of the resolution result. If the global scope does not contain a corresponding symbol, it continues with the top-down inter-model resolution (cf. Section 6.4). The results (i.e., symbols) of this process then are unified with the already found results. Finally, if more than one symbol is found after the resolution request for each potential name finished, the ambiguity will be handled.

While in our approach the name qualification is part of the resolution process, Völkel [Völ11] separates these two phases. There, name qualification is an explicit phase during symbol table creation, as already mentioned in Section 5.1. During this phase each unqualified symbol table entry is replaced by a qualified version (cf. Section 4.1). While this is ok for sequential processing of single models, it hampers a modular and functional manipulation of the scope graph since adding a new symbol requires to explicitly conduct the qualification process.

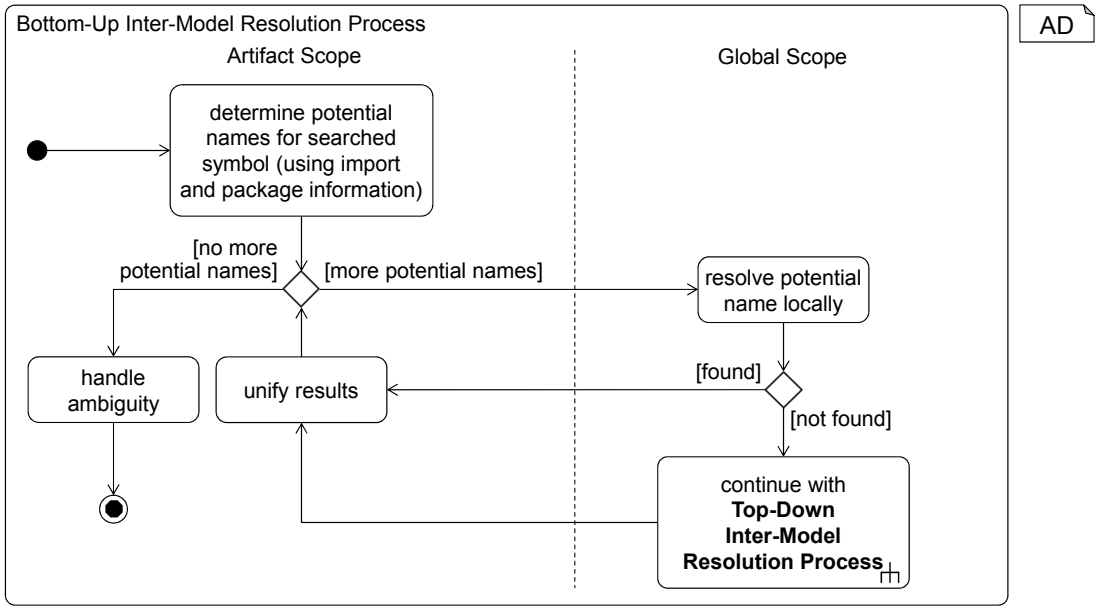


Figure 6.8: General process of the bottom-up inter-model resolution including name qualification. If no symbol is found, the top-down inter-model resolution will continue (cf. Figure 6.11).

6.4 Top-Down Inter-Model Resolution

The global scope is the root of the scope graph (cf. Def. 3.11), and hence, the bottom-up resolution process ends if the global scope is reached (and no symbol is locally found). Instead, the global scope starts the top-down inter-model resolution and continues with its subscopes which—besides scopes of global types—are artifact scopes. In this process, the package declarations of the artifact scopes play an important role.

Figure 6.9 shows an example. The classes C and E both are defined in package p, whereas class D is defined in package k. Consequently, the global scope contains three artifact scopes. Starting the top-down inter-model resolution for $p.E$, the search only continues in artifact scopes declared in package p, namely the artifact scopes of C and E. Within these artifact scopes the package information is not relevant anymore, hence, it is omitted. Consequently, the artifact scope continues with the resolution of E instead of $p.E$. This last step is part of the top-down intra-model resolution process, as described in Section 6.5.

While in the example illustrated in Figure 6.9 it is clear which part of the name embodies the package name (i.e., p), this is not always the case. In Figure 6.10, for example, a symbol with the qualified name $p.q.D.f$ has to be resolved. Given this qualified name, the package name is either $p.q.D$, $p.q$, or p. If the symbol represents a top-level

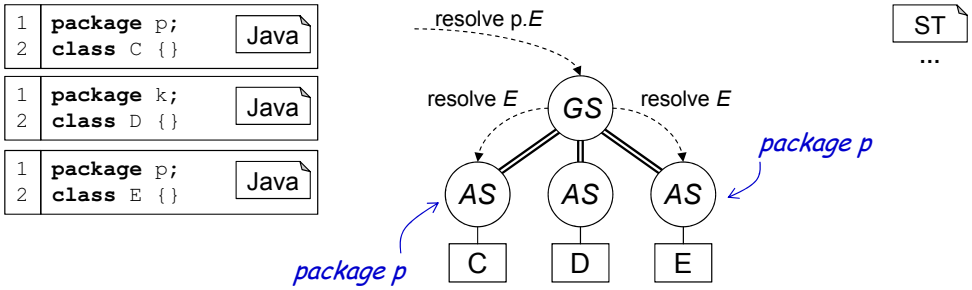


Figure 6.9: Exemplary top-down inter-model resolution via qualified name consisting of two parts. In such a case, the first part is the package name and the second part the symbol's unqualified name.

element, its package name is $p.q.D$. Else, if it represents an inner element (e.g., a field or an inner class), the package name is either p or $p.q$. The resolution process must consider all these cases. Consequently, when resolving $p.q.D.f$, the global scope delegates to each artifact scope defined in one of the above packages. In Figure 6.10, the class C is defined in package p, hence, its artifact scope continues the resolution with $q.D.f$. Analogously, D's artifact scope continues with $D.f$ since its package is $p.q$. Finally, $p.q.D.f$ resolves to the field in D (via a top-down intra-model resolution).

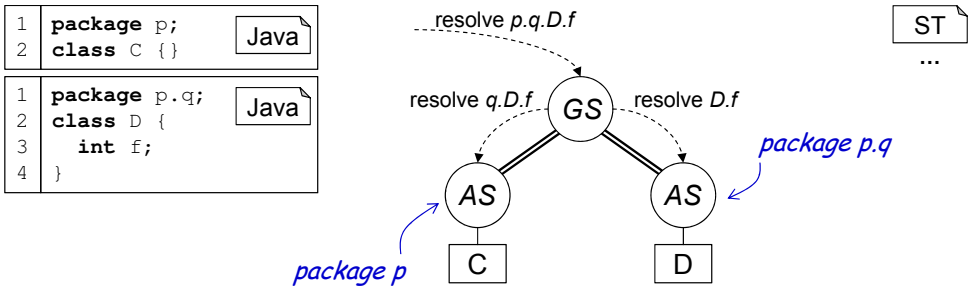


Figure 6.10: Exemplary top-down inter-model resolution via qualified name consisting of more than two parts. In such a case, several possible package names exist.

General Process

As depicted in Figure 6.11, the top-down inter-model resolution process always starts with the global scope. It delegates the resolving request to all artifact scopes, which match the search criteria. This means, the artifact scope's package name must be a prefix of the searched qualified name (which always holds true for the default package). The artifact scope then continues with the top-down intra-model resolution using the remaining part of the name, i.e., without the package prefix (cf. Section 6.5). Similarly,

the top-down intra-model resolution is conducted for each subscope of global scope that is not an artifact scope. Finally, the results of the different resolving processes (i.e., for each subscope) are unified. If more than one symbol is found, the ambiguity will be handled.

Other approaches do not provide an explicit inter-model resolution phase, as in our approach. While in the current thesis the traversal of the scope graph changes (e.g., scope name becomes important instead of shadowing ability) other approaches such as [Völ11, Bet13] allow for programmatically searching in the matching elements in order to access their inner elements. In NaBL [KKWV13] this is realized via *contextual use sites*, which can be specified declaratively.

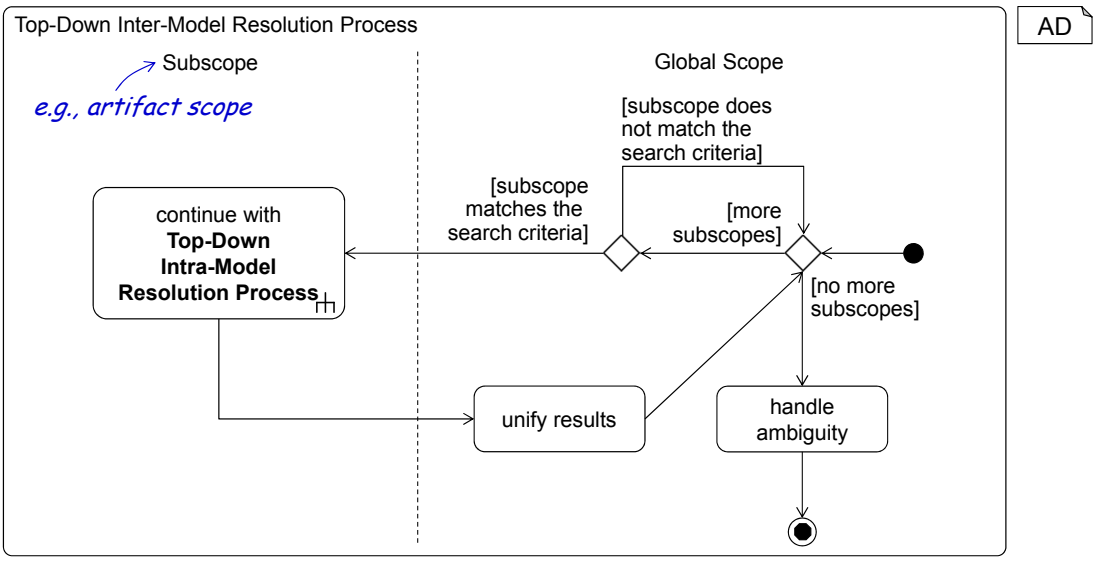


Figure 6.11: General process of the top-down inter-model resolution. If no symbol is found, the top-down intra-model resolution will continue (cf. Figure 6.15).

6.5 Top-Down Intra-Model Resolution

While symbols of enclosing scopes are always visible (if not shadowed) within subscopes (cf. open scope [CL83, GJR79]), the other way around is not necessarily the case. Instead, the subscope must *export* its containing symbols for outside use (cf. Section 3.10). Furthermore, the scope must be named (cf. Def. 3.9) in order to enable access to its symbols via partial (or relative) names [GJR79] (cf. Section 3.2).

Figure 6.12 demonstrates the top-down intra-model resolution process by an example. The outer class C contains the reference $D.f$ which resolves to the field f of the inner class D. This is the case as the referenced name matches the partial name of the field,

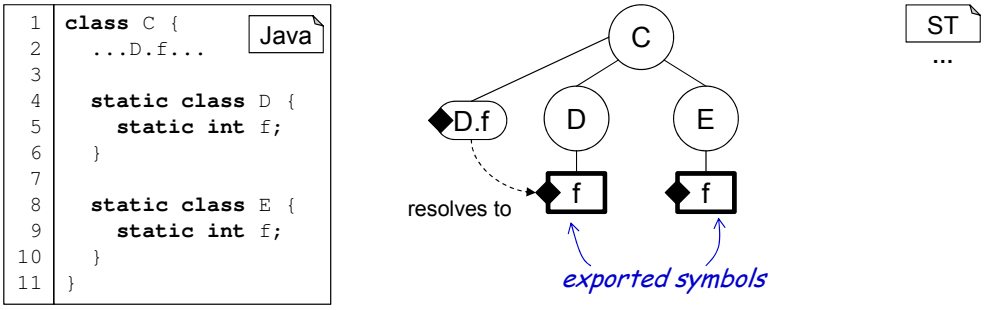


Figure 6.12: Exemplary top-down intra-model resolution where subscopes are named as well as export their contained symbols.

i.e., *D.f*. In order to resolve elements of subscopes, the partial name (or fully qualified name) must match the names of the respective scopes in the scope graph. Figure 6.13 illustrates this by a more complex example.

Starting the resolution from the enclosing scope of A (not shown in Figure 6.13), the resolution continues with A since it matches the first name part of the requested symbol (i.e., $\boldsymbol{A}.B.C.E.f$). Within A, the matching name part is omitted, and hence, A continues with the resolution of $\boldsymbol{B}.C.E.f$, leading to its subscope B. Similarly, the resolution continues with $\boldsymbol{C}.E.f$, and thereafter, with $\boldsymbol{E}.f$. Finally in scope E, the symbol $\boldsymbol{f}$ is found. This only works if all scopes on the path to f yield a matching name, which does not apply to subscope D. Moreover, each subscope on the path must export its containing symbols (cf. Section 3.10). Otherwise, the resolution cannot proceed, as described in the following.

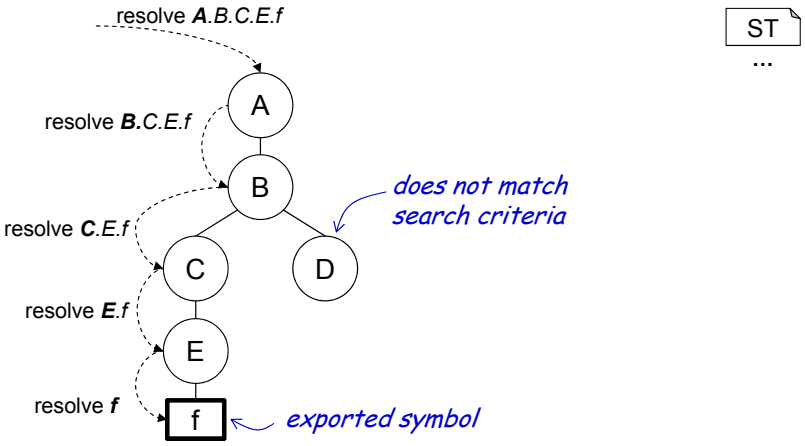


Figure 6.13: Exemplary top-down intra-model resolution in a complex scope graph. Each scope on the path to the requested symbol must have a matching name.

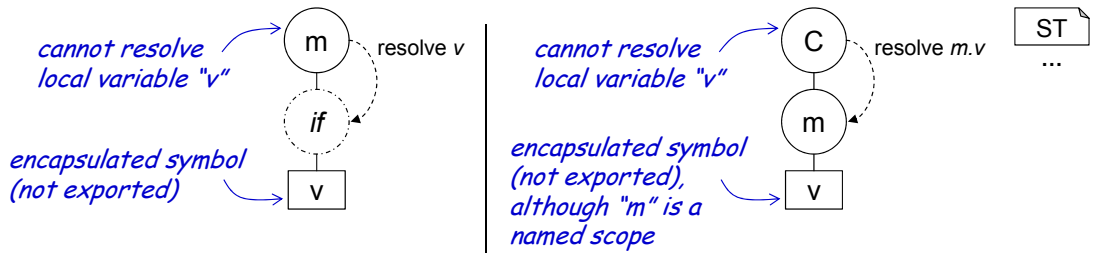


Figure 6.14: Exemplary top-down intra-model resolution in an unnamed scope (left part) and a named scope that does not export its symbols (right part).

In Figure 6.14 (left part) the local variable v is defined in an *if* scope which (in Java and C#) neither is named nor exports its symbols. Hence, there is no possibility for referencing v from outside the *if* scope, e.g., in the enclosing method scope. Similarly, in Figure 6.14 (right part), the local variable is defined in a method scope, and hence, only visible from within the method scope. Otherwise, this could easily lead to ambiguity if methods are overloaded (i.e., same-named but different parameter types). As a consequence, a method scope does not *export* its symbols for outside scopes even though it is named.

In a sum, named scopes by default export their symbols, so that they can be referenced from outside. However, there are exceptions, such as methods in Java. Unnamed scopes do not permit access to their contained symbols. The main reason is that they cannot be referred to unambiguously, which results from the fact that they do not have fully qualified names (cf. Section 3.2).

General Process

Figure 6.15 illustrates the general process for top-down resolution within a model. The process starts with a local search in the current scope. If a matching symbol is found, the resolution ends. Else, if the current scope does not contain any matching symbols, the resolution continues with each subscope that (i) exports its symbols and also (ii) matches the search criteria (e.g., its name matches the requested partial name). Resolved symbols of the different subscopes are unified. Finally, if more than one matching symbol is found, the ambiguity will be handled.

In contrast to the bottom-up approach presented in Section 6.2, the shadowing ability of scopes is ignored in the top-down approach since only symbols of enclosing scopes can be shadowed.

6.6 Resolution in Explicitly Imported Scopes

The previous sections of this chapter mainly focus on *tree-like* scope structures which result from the lexical block hierarchy within artifacts (cf. Section 3.5). Since scopes may

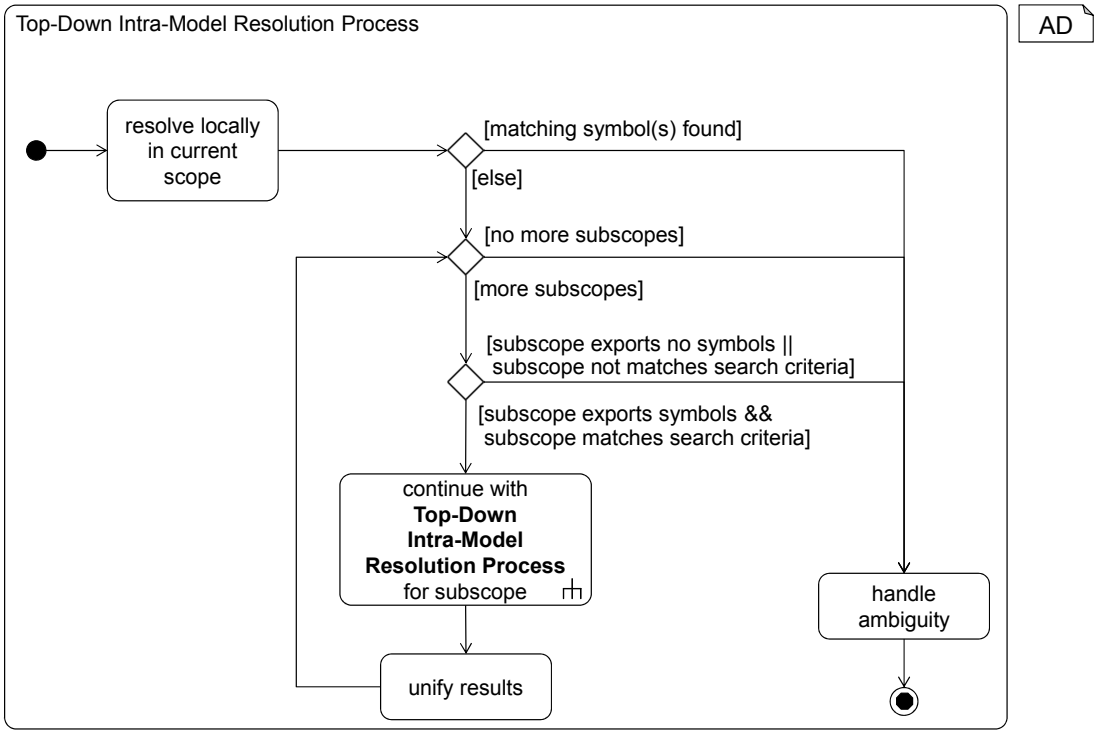


Figure 6.15: General process of the top-down intra-model resolution. The resolution ends here whether or not a symbol is found.

import other scopes—both from the same artifact or other artifacts—the emerging scope graph is not necessarily a tree which complicates the resolution process (cf. [NTVW15]).

As stated in Section 3.10, a scope always *implicitly imports* its enclosing scope. For example, a method scope in Java imports the symbols defined in its enclosing class scope. Furthermore, a scope can *explicitly import* other scopes, e.g., a class scope imports (non-private and not shadowed) symbols of its superclasses and interfaces. Figure 6.16 shows how the resolution is conducted in such a case. Class C extends class S (line 2, upper listing) and refers to the field f (line 3, upper listing) of its superclass. The emerging scope structure is a tree, with the exception that the class scope C imports the scope S. As a result, C imports two scopes, namely the artifact scope (i.e., the lexical enclosing scope) and S (i.e., the explicitly imported scope), and thus, has two enclosing scopes (or, according to Parr [Par10], an enclosing and a parent scope). Therefore, when resolving field f in C, the process can continue with the artifact scope as well as with the scope S. In the current example, f is resolved through the path $C \rightarrow S$.

In cases where both paths lead to a (different) symbol definition, a rule of priority can help solving the ambiguity. In Figure 6.17, class C uses the unqualified type D (line 4, top

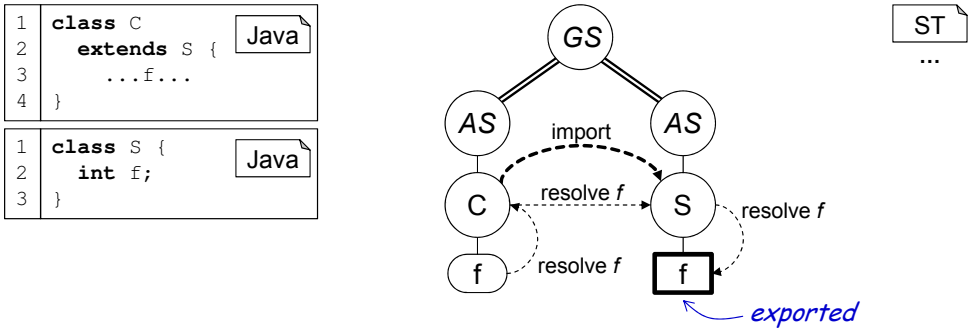


Figure 6.16: Exemplary resolution in an explicitly imported scope.

listing in Figure 6.17). Considering its superclass S, D is an inner class defined in S. The corresponding resolution path is $C \rightarrow S$, marked with *(a)* in the scope graph. Regarding the import statement *p.**, D is the top-level class defined in package p (resulting from the inter-model resolutions described in Section 6.3 and Section 6.4). In this case, the resolution path is $C \rightarrow AS_C \rightarrow GS \rightarrow AS_D$ (marked with *(b)*), where AS_C and AS_D are the artifact scopes of the classes C and D, respectively. Which path has a higher priority depends on a language's semantics. In Java, the superclass has a higher priority than the enclosing artifact scope (cf. [GJS+14]), and hence, D used in C resolves to the inner class of S, i.e., the resolution path is $C \rightarrow S$. Only if D was not defined in S, the resolution path $C \rightarrow AS_C \rightarrow GS \rightarrow AS_D$ would be considered.

As described throughout this chapter, if a symbol is not found in the current scope, the resolution continues with the enclosing scope or the subscopes, depending on whether the bottom-up (cf. Sections 6.2 and 6.3) or the top-down (cf. Sections 6.4 and 6.5) resolution is processing. This, however, does not apply when resolving in explicitly

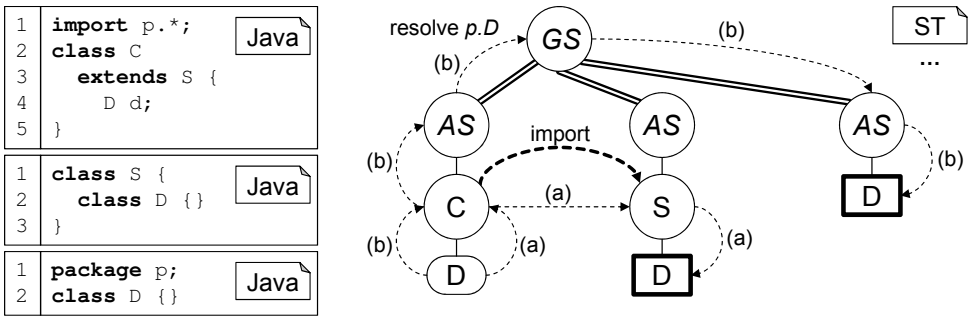


Figure 6.17: Exemplary resolution where several possible paths exist due to an explicitly imported scope. The symbol is resolved unambiguously since explicitly imported scopes have a higher priority than implicitly imported ones.

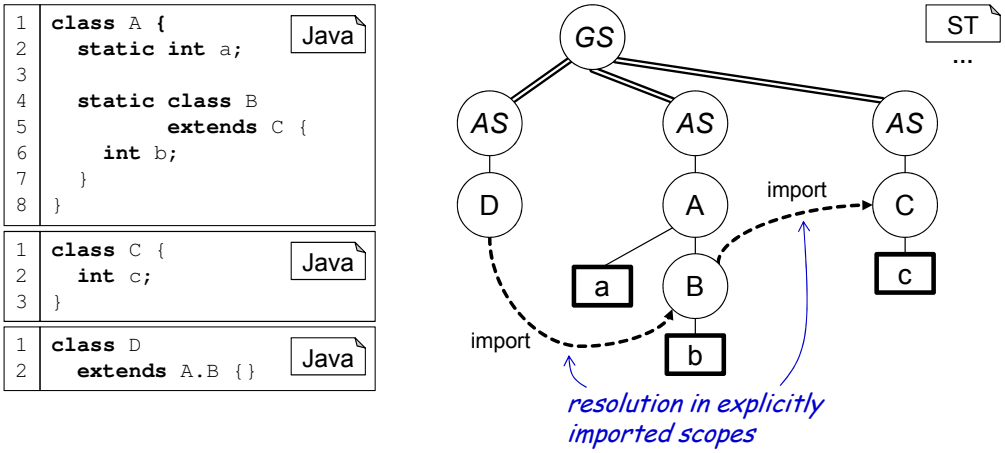


Figure 6.18: Exemplary resolution in an explicitly imported scope ignoring its lexical enclosing scope.

imported scopes [NTVW15]. Figure 6.18 demonstrates this case by an example. Class A defines two members, namely the field a and the static inner class B[3]. Class B subclasses the (top-level) class C and also defines the field b. Finally, class D extends the inner class B.

The right part of Figure 6.18 highlights the emerged scope graph. From that graph it follows that D has access to the fields b and c, imported (transitively) from the scopes B and C, respectively. The corresponding resolution paths started from D are $D \rightarrow B \rightarrow b$ and $D \rightarrow B \rightarrow C \rightarrow c$. The field a defined in class A is not visible in D since B only *implicitly imports* it without exporting it (cf. forwarded symbol Def. 3.19).

In some cases, several resolution paths with the same priority exist. In Figure 6.19, for example, the class scope C imports its superclass' scope S and the interface scope I. Both types define a field f. In Java, elements defined in supertypes have all the same priority. Hence, it is not possible to unambiguously determine the definition of f used in C (line 4, top listing in Figure 6.19). Both paths $C \rightarrow S$ and $C \rightarrow I$ are valid, but lead to a different definition of f.

Since resolution in explicitly imported scopes is highly language-specific, SMI does not provide a default process for it. However, the infrastructure presented in Section 8.5 (and Appendix D) provides reusable default implementations for Java-like languages.

[3]Please note that a static inner class in Java may (only) access static members of its enclosing class [GJS+14]. Hence, in Figure 6.18 the field a (line 2, upper listing) is visible within the inner class B.

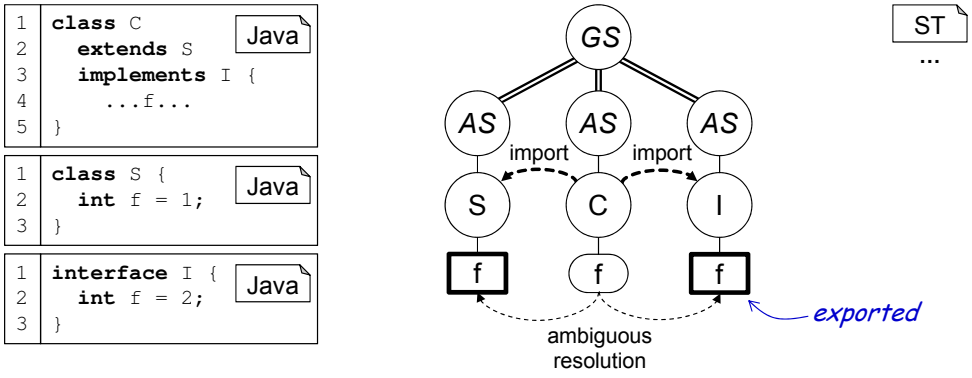


Figure 6.19: Exemplary resolution in several explicitly imported scopes leading to ambiguous results.

6.7 Resolution Using Additional Information

Besides name and kind, some symbols require additional information in order to be unambiguously resolved. Overloaded methods, for example, are both same-named and of the same kind. They are only distinguished by their formal parameters. Hence, those parameters must be specified in the resolution process. Moreover, if a scope does not enable forward references (e.g., a Java method scope), the source position of the reference is important. Then, the resolution process only considers symbols defined (in the model) before the reference.

Furthermore, access modifiers can impact the resolution process. As an example, in order to resolve a *protected* Java *field* symbol named *f*, the specified access modifier for the resolution must be either *private*, *package-local*, or *protected* (or omitted). In contrast, searching for that symbol via the *public* modifier, it will not be resolved—although name and kind match—since *public* does not include *protected* (cf. Section 3.7).

An interesting aspect when resolving via access modifiers is the resolution in explicitly imported scopes, as discussed in Section 6.6. Considering the case depicted in Figure 6.16 (on page 138), the field f is referenced in C. Since f possibly resolves to a field of C, the resolution can start with a *private* access modifier in C. The continuation in C's superclass S, however, cannot continue with *private* but with the *protected* access modifier. Furthermore, if C and S are defined in the same package, the resolution can even continue with the *package-local* access modifier. That way, private fields defined in S are filtered out, and hence, only those fields are considered that are package-local, protected, or public.

The general process for a resolution via additional information is same as the ones introduced throughout this chapter, with one exception: a symbol matches the search criteria if (i) its name and kind match (same as before) and also (ii) the additional information matches (e.g., access modifier or parameter list).

6.8 Technical Infrastructure for the Resolution Mechanism

The resolution process in SMI highly depends on each single scope of the scope graph, e.g., whether it is a shadowing or visibility scope, a Java type scope, the global scope, an artifact scope, etc. To some extend, the scopes are plugged together which results in a composition of scopes (i.e., the scope graph). For this, the local resolution within a scope as described in Section 4.2.2 plays a central role. Similar to Parr [Par10] and MPS [VS10] and different from, among others, Xtext [Bet13], Völkel [Völ11], and Spoofax [KV10], the resolution process in the current thesis is a composition of local resolution processes. That way, scopes can be easily reused and plugged into other scope graphs. The scopes still retain their semantics, e.g., a Java method scope always resolves for parameters as well as local variables (even if applying pattern *(M) Method Spanning a Parameter Scope and a Body Scope* or pattern *(N) Method Spanning an Intermediate Method Scope*, cf. Section 4.3.2). This holds for both, a Java method scope in the Java language itself and a Java method scope that is embedded into another language (cf. Section 8.2).

Technically, the fact that each scope determines how the resolution proceeds next, enables to refer to a scope in a generic way (e.g., `Scope s`), i.e., the type of the scope (e.g., `JavaTypeScope`) is not relevant. This yields the advantage that no type introspection (like `scope instanceof JavaTypeScope`) is required as in [Völ11, Bet13].

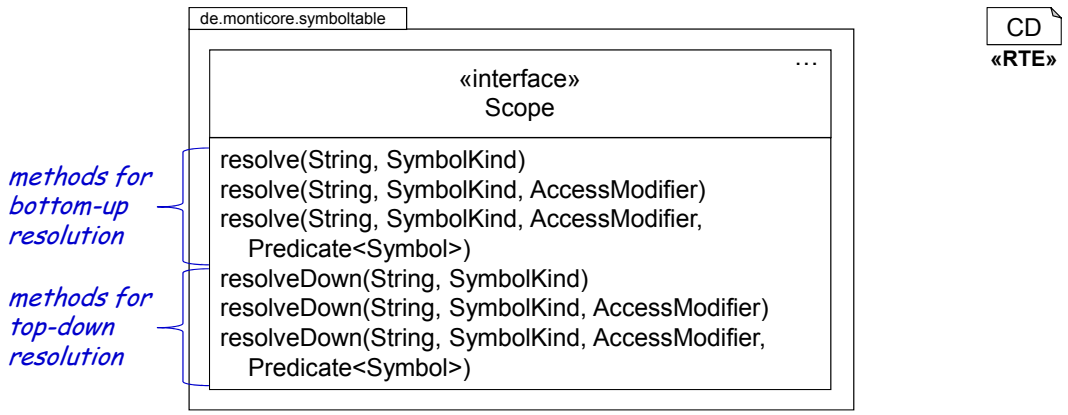


Figure 6.20: Excerpt from the resolution methods provided by the `Scope` interface. These methods *start* the resolution process.

Figure 6.20, Figure 6.21, and Figure 6.22 highlight the essential methods of the `Scope` hierarchy that participate in the resolution process (cf. overview of technical scope classes in Figure 4.16 on page 69). As mentioned in Section 4.2, the `Scope` interface first of all serves the language user, and hence, provides methods for resolving symbols from the scope graph. In contrast, `MutableScope`—which extends `Scope`—is meant for the language engineer who needs to modify the scope graph (e.g., during the symbol table

creation, cf. Chapter 5). Furthermore, it provides methods for the internal resolution process—which is hidden away from the language user—like continuing with the enclosing scope if no scope was found in the current scope. Finally, the `CommonScope` is a technical class that provides reasonable defaults for both `Scope` and `MutableScope` (cf. *RRQ1*), based on the resolution process as described previously in this chapter.

The following gives an overview of `Scope`'s methods presented in Figure 6.20. These methods are partially used in several resolution phases and either serve the bottom-up or the top-down resolution. The methods for the bottom-up resolution phases (cf. Section 6.2 and Section 6.3) are:

resolve(String, SymbolKind) Starts the *bottom-up* resolution process with the given symbol name (first parameter) and kind (second parameter). Depending on the scope, it starts the intra-model resolution process (cf. Section 6.2) or the inter-model resolution process (cf. Section 6.3). This method expects only one matching symbol, and hence, it standardly handles the ambiguity (cf. *"handle ambiguity"* in Figure 6.6 and Figure 6.8) by throwing an exception. Classes implementing this method can choose alternative implementations, e.g., returning the first matching symbol.

resolve(String, SymbolKind, AccessModifier) Following Section 6.7, the resolution process via access modifiers is conducted the same way as without modifiers. Consequently, this method behaves same as `resolve(String, SymbolKind)` and additionally filters out symbols with access modifiers that are not included (cf. Def. 3.14) in the specified modifier (third parameter).

resolve(String, SymbolKind, AccessModifier, Predicate<Symbol>) Besides the three parameters as in the previous method, this method allows for specifying further conditions (fourth parameter) the searched symbol must fulfill. This enables, for example, to specify the parameter list of a method in order to distinguish it from other overloaded methods (cf. Section 6.7).

Additionally, `Scope` provides a `resolveMany` method for each of the above listed methods, such as `resolveMany(String,SymbolKind)` (not shown in Figure 6.20). The only difference is that a `resolveMany` method always returns a collection of the resolved symbols, and hence, no exception is thrown. Context conditions (cf. Section 2.2.5) can employ these methods to check the well-formedness of a model.

Analogously, the methods for the top-down resolution phases (cf. Section 6.5 and Section 6.4) provided by `Scope` are:

resolveDown(String, SymbolKind) Starts the *top-down* resolution process with the given symbol name (first parameter) and kind (second parameter). Depending on the scope, it starts the intra-model resolution process (cf. Section 6.5) or the

inter-model resolution process (cf. Section 6.4). This method expects only one matching symbol, and hence, it standardly handles the ambiguity (cf. *"handle ambiguity"* in Figure 6.11 and Figure 6.15) by throwing an exception. Classes implementing this method can choose alternative implementations, like returning the first matching symbol.

resolveDown(String, SymbolKind, AccessModifier) Behaves the same as `resolveDown(String, SymbolKind)` and additionally filters out symbols having access modifiers that are not included (cf. Def. 3.14) in the specified modifier (third parameter).

resolveDown(String, SymbolKind, AccessModifier, Predicate<Symbol>) Besides the three parameters as in the previous method, this method allows for specifying further conditions (fourth parameter) the searched symbol must fulfill. This enables, for example, to specify the parameter list of a method in order to distinguish it from other overloaded methods (cf. Section 6.7).

Similar to the methods of the bottom-up resolution process, for each of the above listed methods a `resolveDownMany` method exists which returns a collection of the resolved symbols and does not throw an exception.

The methods of the `Scope` interface *start* the respective resolution process beginning with the current scope. If the resolution must *continue* with other scopes (e.g., because no symbol was found), sometimes information about the resolution process is required. For example, in Figure 6.5, the resolution ends after the method scope is processed since a field has already been found in the method's subscope. Such information is stored in `ResolvingInfo`. For this, the `MutableScope` interface—as highlighted in Figure 6.21—provides methods which serve the *automatic continuation* of the resolution process. These methods are:

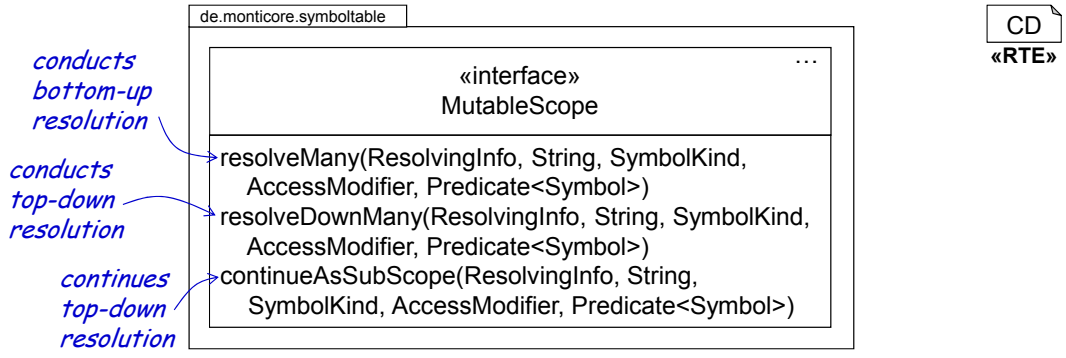


Figure 6.21: Excerpt from the resolution methods provided by the `MutableScope` interface. These methods *continue* an already started resolution process.

resolveMany(ResolvingInfo, ..., Predicate<Symbol>) Same as the method `resolveMany(String, SymbolKind, AccessModifier, Predicate<Symbol>)` of the `Scope` interface, this method conducts a bottom-up resolution. It additionally contains important information for the resolution process (first parameter), such as whether symbols have already been found and which resolving filters are registered (cf. Section 4.2.2).

resolveDownMany(ResolvingInfo, ..., Predicate<Symbol>) This method conducts the top-down resolution process as described for `resolveDownMany(String, SymbolKind, AccessModifier, Predicate<Symbol>)` and additionally provides useful information of the resolution process via its first parameter.

continueAsSubScope(ResolvingInfo, ..., Predicate<Symbol>) This method is invoked on each subscope during the top-down resolution process. Each subscope then determines itself whether it fulfills the conditions (e.g., exports symbols) for continuing with the top-down resolution.

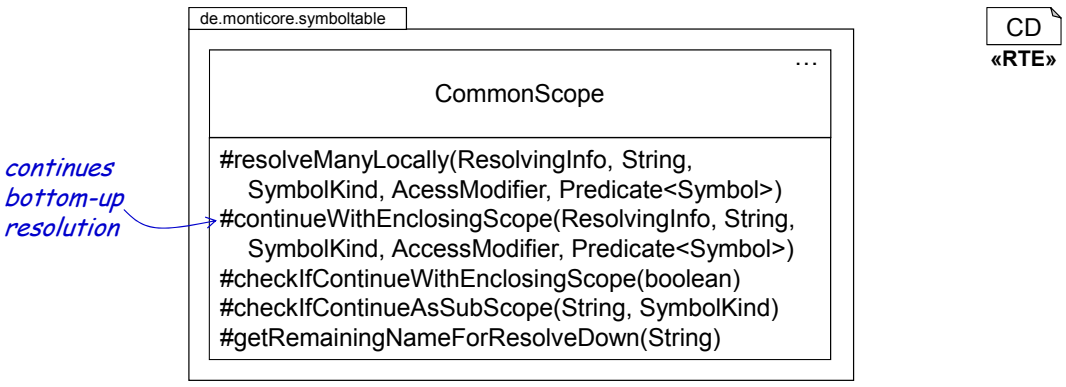


Figure 6.22: Excerpt from the resolution methods provided by the `CommonScope` class. These methods serve as hook methods for language-specific customizations.

In addition to the methods of `Scope` and `MutableScope`, the `CommonScope` class depicted in Figure 6.22 provides some hook methods [Pre95a] that can be overridden by language-specific scopes to easily customize the resolution process (cf. *RRQ2*). Please note that this methods are neither part of `Scope` nor `MutableScope` since they are not required as interface between scopes. In other words, they only serve to control the resolution *within the current scope*, while methods such as `resolve(String, SymbolKind)` are also employed to conduct the resolution process *in other scopes*.

resolveManyLocally(ResolvingInfo, ..., Predicate<Symbol>) Resolves all matching symbols locally found in the current scope.

continueWithEnclosingScope(ResolvingInfo, ..., Predicate<Symbol>) Continues the bottom-up resolution with the enclosing scope, if method `checkIfContinueWithEnclosingScope` returns true (see next).

checkIfContinueWithEnclosingScope(boolean) Checks whether the resolution should continue with the enclosing scope, for example, if no symbols are found or the current scope is no shadowing scope.

checkIfContinueAsSubScope(String, SymbolKind) Checks whether the current scope should continue as subscope with the top-down resolution. By default, this method is called in the implementation of `MutableScope`'s method `continueAsSubScope(ResolvingInfo, String, SymbolKind, AccessModifier)`.

getRemainingNameForResolveDown(String) As described in Sections 6.4 and 6.5, the top-down resolution continues with a part of the requested symbol name, which is determined by this method.

Please note that the method descriptions do not distinguish between intra-model and inter-model resolution since this depends on the scope (type) itself. For example, a Java class scope conducts an intra-model resolution while the global scope conducts an inter-model resolution.

The `Scope` interface provides many similar methods which, for example, only differ in how ambiguity is handled. While this simplifies the work of the language user, it can hamper the work of the language engineer since a lot of redundancy exists. For example, customizing the bottom-up resolution might affect all of the first three methods of the `Scope` interface shown in Figure 6.20. However, as presented in the activity diagrams for the general processes (i.e., Figures 6.6, 6.8, 6.11, and 6.15) the ambiguity handling is always the last activity, and hence, does not affect the previous process. Also, resolving via additional information (such as access modifiers, cf. Section 6.7) does not influence the previous resolution process but rather restricts what "matching symbols" are. Following from these observations, `CommonScope` implements the resolution logic *only* in its own methods and the methods it inherits from `MutableScope`. All methods inherited from `Scope` *delegate* to those methods. As a consequence, only a few methods must be overridden in order to customize the resolution, following the *narrow inheritance interface principle* [WGM89, Pre94] (cf. *RRQ2*).

Figure 6.23 shows the call-stack starting with the `resolve(String, SymbolKind)` method (analogously for `resolveDown(String, SymbolKind)`). First, `resolve(String, SymbolKind)` delegates to the method `resolveMany(String, SymbolKind)` which itself delegates to `resolveMany(String, SymbolKind, AccessModifier)` (not shown in Figure 6.23). For this, the `ALL_INCLUSION` constant of `AccessModifier` is employed, which includes any modifier (cf. Section 4.5), and hence, does not restrict the resolution. Next, `resolveMany(String, SymbolKind,`

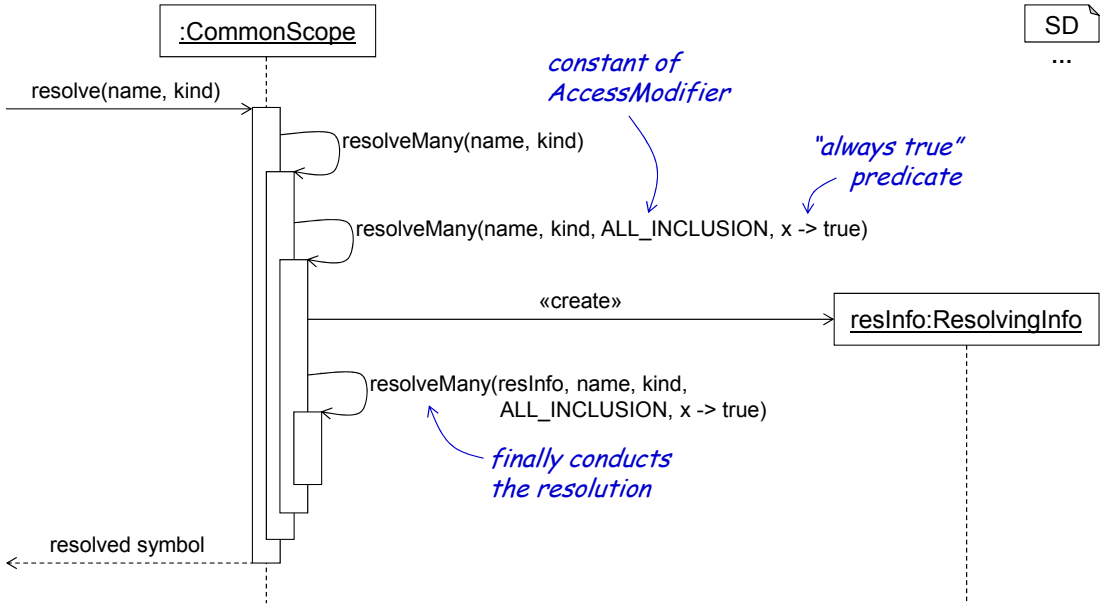


Figure 6.23: Delegation procedure of the resolution methods as implemented in the `CommonScope` class. Method `resolveMany(ResolvingInfo, ..., Predicate<Symbol>)` ultimately conducts the resolution process.

`AccessModifier, Predicate<Symbol)` is called with a predicate that is always true as fourth parameter. Same as for `ALL_INCLUSION`, the predicate does not restrict the resolution (since always true). `resolveMany(String, SymbolKind, AccessModifier, Predicate<Symbol)` then instantiates `ResolvingInfo` and further delegates to `resolveMany(ResolvingInfo, ..., Predicate<Symbol>)` which ultimately conducts the resolution. The results, i.e., the resolved symbols, are finally passed to the initially called `resolve(String, SymbolKind)` method. If more than one symbol is found, the method throws an exception, else it returns the result. In contrast, directly starting the resolution via `resolveMany(String, Symbol)`, returns all matching symbols.

Besides reducing redundancy throughout the methods and preventing inconsistencies between them, the delegation approach has the advantage that the language engineer only has to adjust one method in order to customize the resolution process of a scope for language-specific needs. The remainder of this section presents technical realizations for major aspects of the resolution phases.

6.8.1 Technical Realization of Bottom-Up Intra-Model Resolution

The different `resolve(...)` as well as `resolveMany(...)` methods provided by `CommonScope` start the bottom-up intra-model resolution. Each of these methods ultimately delegates to `resolveMany(ResolvingInfo, ..., Predicate<Symbol>)` (cf. Figure 6.23) which realizes the different activities of the resolution process presented in Figure 6.6 (on page 129) as shown in Listing 6.24.

Java «RTE»

```
1 @Override
2 public <T extends Symbol> Collection<T> resolveMany(
3     ResolvingInfo resInfo, String name, SymbolKind kind,
4     AccessModifier modifier, Predicate<Symbol> predicate) {
5
6   // Resolve symbol locally in the current scope
7   Set<T> resolvedSymbols =
8     resolveManyLocally(resInfo, name, kind, modifier, predicate);
9
10  // Resolve symbol in the enclosing scope
11  Collection<T> resFromEnclosing = continueWithEnclosingScope
12    (resInfo, name, kind, modifier, predicate);
13
14  // Unify results of current scope and its enclosing scope(s)
15  resolvedSymbols.addAll(resFromEnclosing);
16
17  return resolvedSymbols;
18 }
```

Listing 6.24: Implementation of method `resolveMany(ResolvingInfo, ..., Predicate<Symbol>)` of class `CommonScope`.

1. **Local search:** First, a local search within the current scope is conducted (lines 7–8, Listing 6.24). This corresponds to the *"resolve locally in current scope"* activity depicted in Figure 6.6.

2. **Continuation in enclosing scope:** Next, the method `continueWithEnclosingScope` of `CommonScope` (lines 11–12) eventually continues the resolution with the enclosing scope. Listing 6.25 shows the implementation of `continueWithEnclosingScope`. First, it checks whether the resolution should be continued with the enclosing scope via `checkIfContinueWithEnclosingScope`. By default, this is the case if no symbols have been found so far or the scope is a visibility scope (line 3, Listing 6.26), which corresponds to the first condition of Figure 6.6. If the check is true and an enclosing scope exists (lines 6–7 in Listing 6.25, also see second condition of Figure 6.6), the resolution continues with the enclosing scope

(lines 8–9). Please note that in contrast to the activity diagram in Figure 6.6 there is no check whether the current scope is an artifact scope or not. Instead, this is conducted via polymorphism by invoking its `resolveMany(ResolvingInfo, ..., Predicate<Symbol>)` method.

3. **Unifying results:** The last step unifies the symbols resolved in the current scope and the ones resolved in its enclosing scope (line 15, Listing 6.24). This corresponds to the *"unify results"* activity in Figure 6.6.

4. **Handle Ambiguity:** Finally, if several symbols are resolved, an exception will be thrown in case the resolution was started via a `resolve(...)` method (cf. *"handle ambiguity"* activity in Figure 6.6).

```
1 protected                                                          Java
2 <T extends Symbol> Collection<T> continueWithEnclosingScope(      «RTE»
3     ResolvingInfo resInfo, String name, SymbolKind kind,
4     AccessModifier modifier, Predicate<Symbol> predicate) {
5
6   if (checkIfContinueWithEnclosingScope(resInfo.areSymbolsFound())
7       && (getEnclosingScope().isPresent())) {
8     return getEnclosingScope().get()
9         .resolveMany(resInfo, name, kind, modifier, predicate);
10  }
11
12  return Collections.emptySet();
13 }
```

Listing 6.25: Implementation of method `continueWithEnclosingScope` of class `CommonScope`.

```
1 protected                                                          Java
2 boolean checkIfContinueWithEnclosingScope(boolean found) {        «RTE»
3   return !(found && isShadowingScope());
4 }
```

Listing 6.26: Implementation of method `checkIfContinueWithEnclosingScope` of class `CommonScope`.

6.8.2 Technical Realization of Bottom-Up Inter-Model Resolution

The artifact scope's first steps of the inter-model resolution process are the same as the default for other scopes. The difference begins as soon as the resolution con-

tinues with the enclosing scope since then the artifact scope starts the bottom-up inter-model resolution process, as depicted in Figure 6.8 (on page 132). Consequently, the `ArtifactScope` class completely reuses the `resolveMany(ResolvingInfo, ..., Predicate<Symbol>)` method inherited from `CommonScope` and only overrides the method `continueWithEnclosingScope(ResolvingInfo, ..., Predicate<Symbol>)`, as shown in Listing 6.27.

The difference to the `continueWithEnclosingScope` method of `CommonScope` (cf. Listing 6.25) is the for-loop (lines 14–19, Listing 6.27). Here, instead of solely continuing the resolution with the enclosing scope, the `ArtifactScope` determines all potential names of the searched symbol via a `QualifiedNamesCalculator` (cf. *"determine potential names for searched symbol (...)"* activity in Figure 6.8). Its method `calculateQualifiedNames` (line 12) considers all possible names discussed in Section 6.3. Subsequently, for each of those names the `resolveMany(ResolvingInfo, ..., Predicate<Symbol>)` method of the enclosing scope (i.e., the global scope) is invoked (lines 15–17, Listing 6.27). Please note that `ArtifactScope` does not need to override `checkIfContinueWithEnclosingScope` since the criteria are the same as the default implemented in Listing 6.26. Same holds true for the unification of the results.

Since the global scope cannot resolve upwards, the `GlobalScope` class overrides the `resolveMany(ResolvingInfo, ..., Predicate<Symbol>)` method in order to delegate to `resolveDownMany(ResolvingInfo, ..., Predicate<Symbol>)`. That way, `GlobalScope` starts the top-down inter-model resolution (cf. *"continue with Top-Down Inter-Model Resolution Process"* activity in Figure 6.8). Thanks to the delegations depicted in Figure 6.23, this affects any method in `GlobalScope` concerning the bottom-up resolution, e.g., `resolve(String, SymbolKind)`. Listing 6.28 shows an excerpt from the `GlobalScope`'s `resolveMany` method. As it can be seen, `resolveMany` directly starts the top-down process (lines 7–8) by delegating to (`CommonScope`'s) method `resolveDownMany(ResolvingInfo, ..., Predicate<Symbol>)`.

6.8.3 Technical Realization of Top-Down Inter-Model Resolution

The global scope is never in the role of a subscope. Therefore, the `GlobalScope` class overrides the method `checkIfContinueAsSubScope` in order to always return `false`. As a consequence, invoking `continueAsSubScope` on the `GlobalScope` always results in an empty collection. Furthermore, the global scope can contain scopes of global symbols as well as artifact scopes. In the former case, the continuation of the top-down resolution process is the same as described in Section 6.8.4. In the latter case, however, the search criteria as described in Section 6.4 must be checked. For this, the `ArtifactScope` class overrides `checkIfContinueAsSubScope` and adjusts the check as described in Section 6.4. Listing 6.29 shows the implementation of its `checkIfContinueAsSubScope` method. First, it is checked whether the scope

```java
@Override
protected
  <T extends Symbol> Collection<T> continueWithEnclosingScope(
     ResolvingInfo resInfo, String name, SymbolKind kind,
     AccessModifier modifier, Predicate<Symbol> predicate) {

  Collection<T> result = new LinkedHashSet<>();
  if (checkIfContinueWithEnclosingScope(resInfo.areSymbolsFound())
      && (getEnclosingScope().isPresent())) {

    Set<String> potentialQualifiedNames = qualifiedNamesCalculator
        .calculateQualifiedNames(name, packageName, imports);

    for (String qualifiedName : potentialQualifiedNames) {
      Collection<T> resFromEnclosing = getEnclosingScope().get()
          .resolveMany(resInfo, qualifiedName,
              kind, modifier, predicate);
      result.addAll(resFromEnclosing);
    }
  }
  return result;
}
```

Java «RTE»

Listing 6.27: Implementation of method `continueWithEnclosingScope` of class `ArtifactScope`.

```java
@Override
public <T extends Symbol> Collection<T> resolveMany(
   ResolvingInfo resolvingInfo,
   String name, SymbolKind kind,
   AccessModifier modifier, Predicate<Symbol> predicate) {

  Collection<T> resolvedSymbol = resolveDownMany(
          resolvingInfo, name, kind, modifier, predicate);

  // ...

  return resolvedSymbol;
}
```

Java «RTE»

Listing 6.28: Excerpt from method `resolveMany(ResolvingInfo, ..., Predicate<Symbol>)` of class `GlobalScope`.

exports its symbols (line 5, Listing 6.29), which by default holds true for artifact scopes. Otherwise, the method finishes and returns `false` (line 34), i.e., the artifact scope will not continue the resolution.

Java «RTE»

```
1 @Override
2 protected boolean checkIfContinueAsSubScope(
3     String name, SymbolKind kind) {
4
5   if(this.exportsSymbols()) {
6     String symbolQualifier = Names.getQualifier(name);
7
8     List<String> symbolQualifierParts =
9         Splitters.DOT.splitToList(symbolQualifier);
10    List<String> packageParts =
11        Splitters.DOT.splitToList(packageName);
12
13    boolean symbolNameStartsWithPackage = true;
14
15    if (packageName.isEmpty()) {
16      // symbol qualifier always contains the
17      // default package (i.e., empty string)
18      symbolNameStartsWithPackage = true;
19    }
20    else if (symbolQualifierParts.size() >= packageParts.size()) {
21      for (int i = 0; i < packageParts.size(); i++) {
22        if (!packageParts.get(i)
23            .equals(symbolQualifierParts.get(i))) {
24          symbolNameStartsWithPackage = false;
25          break;
26        }
27      }
28    }
29    else {
30      symbolNameStartsWithPackage = false;
31    }
32    return symbolNameStartsWithPackage;
33  }
34  return false;
35 }
```

Listing 6.29: Implementation of method `checkIfContinueAsSubScope` of class `ArtifactScope`.

The artifact scope only continues with the resolution if its package name is a prefix of the symbol's qualifier, i.e., the fully qualified name without the symbol's simple name.

This always holds true for the default package whose name technically is an empty string (lines 15–19). For non-default packages, additional checks are required; the helper class `Splitters` is employed which creates a list with elements for each part of a dot-separated name. For `"java.lang"`, for example, a list with the two elements `"java"` and `"lang"` is created. This is done for the symbol's qualifier (lines 8–9) and the artifact scope's package name (10–11) stored in the local variables `symbolQualifierParts` and `packageParts`, respectively. Next, lines 20–28 check (i) that the symbol qualifier's name parts are not fewer than the package's name parts and (ii) that each part in the package equals the parts in the symbol's qualifier at the same position. If one of the conditions is not fulfilled, the variable `symbolNameStartsWithPackage` is set to `false` (lines 24 and 30, Listing 6.29), hence, the artifact scope will not proceed with the resolution. This check ensures that the resolution continues in cases like presented in Figure 6.9 and Figure 6.10. `ArtifactScope` also overrides `CommonScope`'s method `getRemainingNameForResolveDown` to calculate the remaining name of the searched symbol as described in Section 6.4.

Please note that the `GlobalScope` does not know the specific types of its subscopes. This is possible since the subscopes decide themselves when to continue the top-down resolution by overriding `CommonScope`'s method `checkIfContinueAsSubScope` and eventually the method `continueAsSubScope`. Finally, the subscope—whether artifact scope or not—continues with the top-down intra-model resolution via its `resolveDownMany` method as described next.

6.8.4 Technical Realization of Top-Down Intra-Model Resolution

Analogously to Section 6.8.1, the `resolveDown(...)` and `resolveDownMany(...)` methods provided by `CommonScope` start the top-down intra-model resolution process. For this, they delegate to `resolveDownMany(ResolvingInfo, ..., Predicate<Symbol>)`. Listing 6.30 shows `CommonScope`'s implementation of this method which realizes the resolution process depicted in Figure 6.15.

1. **Local search:** First, `resolveDownMany` conducts a local search (lines 8–9) within the current scope (cf. *"resolve locally in current scope"* activity in Figure 6.15).

2. **Continuation with subscopes:** If no matching symbols are found in the current scope (first condition in Figure 6.15), the resolution will continue with each subscope (second condition in Figure 6.15), as stated in lines 11–21 of Listing 6.30. For this, the `continueAsSubScope` method of each subscope is invoked (lines 15–16, Listing 6.30). The subscope first checks whether it fulfills the criteria (third condition, Figure 6.15) for continuing with the top-down resolution (line 6, Listing 6.31) which is implemented in method `checkIfContinueAsSubScope`. The condition is true if the subscope exports its symbols (line 4, Listing 6.32) and also matches the naming criteria as described in Section 6.5. Technically, the name must consist

```java
@Override
public
  <T extends Symbol> Collection<T> resolveDownMany(
        ResolvingInfo resInfo, String name, SymbolKind kind,
        AccessModifier modifier, Predicate<Symbol> predicate) {

  // Conduct search locally in the current scope
  Set<T> resolved =
    resolveManyLocally(resInfo, name, kind, modifier, predicate);

  if (resolved.isEmpty()) {
    // 2. Continue search in sub scopes and ...
    for (MutableScope subScope : getSubScopes()) {
      Collection<T> resolvedFromSub =
       subScope.continueAsSubScope(resInfo, name, kind,
          modifier, predicate);

      // 3. ...unify results
      resolved.addAll(resolvedFromSub);
    }
  }

  return resolved;
}
```

Java «RTE»

Listing 6.30: Implementation of method `resolveDownMany(ResolvingInfo, ..., Predicate<Symbol>)` of class `CommonScope`.

of at least two parts $p.N$ (line 7, Listing 6.32). The name of the scope must be equal to the first part p of the searched name. Only if the subscope matches the search criteria, it continues the top-down resolution (lines 10–11, Listing 6.31) with the remaining name, determined by `getRemainingNameForResolveDown` (lines 7–8, Listing 6.31).

In a sum, `continueAsSubScope` implements the third condition as well as the *"continue with Top-Down Intra-Model Resolution Process for subscope"* activity depicted in Figure 6.15.

3. **Unifying results:** The results of the current scope and the subscopes are iteratively unified (cf. *"unify results"* activity in Figure 6.15) in the for-loop (line 19, Listing 6.30).

Please note that not the *current scope* determines whether a subscope matches the search criteria (cf. third condition in Figure 6.15)—and hence, should continue with the

```java
@Override
public <T extends Symbol> Collection<T> continueAsSubScope(
        ResolvingInfo resInfo, String name, SymbolKind kind,
        AccessModifier modifier, Predicate<Symbol> predicate) {

  if (checkIfContinueAsSubScope(name, kind)) {
    String remainingName =
      getRemainingNameForResolveDown(name);

    return this.resolveDownMany
      (resInfo, remainingName, kind, modifier, predicate);
  }
  return Collections.emptySet();
}
```

Java «RTE»

Listing 6.31: Implementation of method `continueAsSubScope` of class `CommonScope`.

```java
protected boolean checkIfContinueAsSubScope
  (String name, SymbolKind kind) {

  if(this.exportsSymbols()) {
    List<String> nameParts = getNameParts(name).toList();

    if (nameParts.size() > 1) {
      String firstNamePart = nameParts.get(0);
      return firstNamePart.equals(getName().orElse(""));
    }
  }
  return false;
}
```

Java «RTE»

Listing 6.32: Implementation of method `checkIfContinueAsSubScope` of class `CommonScope`.

search—but the *subscope* itself. Otherwise, the current scope would have to know about the concrete type of its subscopes in order to determine whether they fulfill the search criteria. This not only requires type introspection but also hampers language embedding where subscopes of other languages can be embedded into a scope (cf. Section 8.2). Certainly, the types of those subscopes are unknown in the embedding scope. In contrast, during the bottom-up resolution (cf. Section 6.8.1) the *current scope* determines whether to continue with its enclosing scope, and not the *enclosing scope* itself. This is because

the bottom-up resolution depends on the criteria of the *current* scope (e.g., whether it is a shadowing scope), and is independent from the (type of the) enclosing scope. Consequently, this also works for language embedding.

6.8.5 Technical Realization of Resolution in Explicitly Imported Scopes

To distinguish resolving in explicitly imported scopes from resolving in lexical enclosing scope(s) (cf. Section 3.5), the `Scope` interface provides the method `resolveImported(String, SymbolKind, AccessModifier, Predicate<Symbol>)`. The default implementation provided by `CommonScope` only conducts a local search in the current scope. This is because the resolution in imported scopes is highly language-specific, as already stated in Section 6.6.

6.9 Model Loading

Up to now, the *entire* scope graph has always been considered. That means, each model of the model path is already processed, and hence, in the memory. While this is ok for a small number of models, it can lead to efficiency issues if a great amount of models exists. Therefore, SMI standardly loads and processes models on demand, when requested. This ensures that only models that are used are loaded and all others are not. Figure 6.33 shows an example.

The class C extends the class D. The corresponding scope tree in Figure 6.33 (a) highlights that only class C is loaded while the file *D.java* in the model path is not. To validate C, it, among others, has to be ensured that D exists and is a non-final class. This requires to load the scope graph of D and add it to the global scope. From then on, its information can be resolved as usual. The global scope (in combination with the languages model loader, cf. Section 6.9.3) is responsible for loading (and processing) the *D.java* file and building up the corresponding scope graph (cf. Chapter 5, as shown in Figure 6.33 (b)).

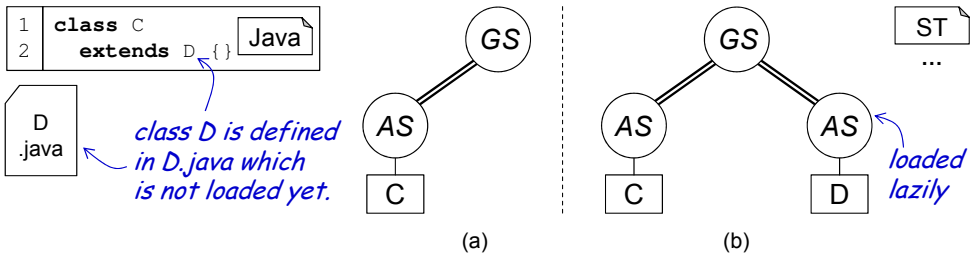


Figure 6.33: Example of lazy model loading.

To efficiently find the physical location that corresponds to a logical name, languages, such as Java and MontiArc [HRR12], identify a type (Java) or a component (MontiArc) via the file name. Moreover, packages are identified via (sub-)directories. For example, a Java class C in a package p is stored in a *C.java* file in a directory *p*. When searching for the logical name $p.C$ the class loader considers the directories of the class path that are named *p*. This among others speeds up the search process and also simplifies the manual search since the logical name can be easily matched to the physical name. In contrast, in languages like C, the `include` statement only refers to the file name.

While the Java approach works for top-level elements, only a part of the qualified name of inner model elements (such as fields and methods in Java) describes the physical location. For example, a field f of class C in package p, is stored in file *p/C.java*. Consequently, the scope graph of the class $p.C$ must be loaded into the global scope. After that, f can be resolved in $p.C$. Hence, the logical name $p.C.f$ maps to the physical name *p/C.java* instead of *p/C/f.java*.

Furthermore, some symbol kinds represent both top-level and inner elements. In Java, *top-level* classes as well as *inner* classes exist. Resolving a class $p.C.D$, does not specify whether D is a top-level or an inner class (only the Java naming convention may hint to it). Consequently, the model's physical location is either $p/C/D.java$ or $p/C.java$. If both locations exist, the model cannot be resolved unambiguously. Since inner classes can contain inner classes themselves, the search can become even deeper.

In a sum, when searching for a top-level symbol via its qualified name $n_1.n_2...n_{k-1}.n_k$, the corresponding model's physical location is $n_1/n_2/.../n_{k-1}/n_k.ext$, where ext is the language's file extension. Hence, the model's and the symbol's qualified name are equal. In contrast, searching for an inner symbol such as an inner class, the model's location is $n_1/n_2/.../n_{k-1}.ext$, $n_1/n_2/.../n_{k-2}.ext$, ..., or $n_1.ext$, depending on whether the enclosing symbol is a top-level symbol or an inner symbol itself.

6.9.1 General Process

The global scope enables references between models via the top-down inter-model resolution process (cf. Section 6.4). In order to (lazily) load models during resolution, the default top-down inter-model resolution process is extended. As shown in Figure 6.34, the model loading process is an integrated part of the resolution process, and thus, encapsulated from the language user (cf. *RRQ4*).

First, a top-down inter-model resolution is conducted on the scope graph, as described in Section 6.4, which ends, if one or more matching symbols are found. Otherwise, it is not clear whether the requested symbol does not exist or whether its containing model is not loaded yet[4]. Hence, in the next step, all models that match the naming convention of the requested symbol (as described above), are loaded and added to the global scope.

[4]To increase efficiency, the resolution mechanism only tries to load a matching model when it is requested for the first time.

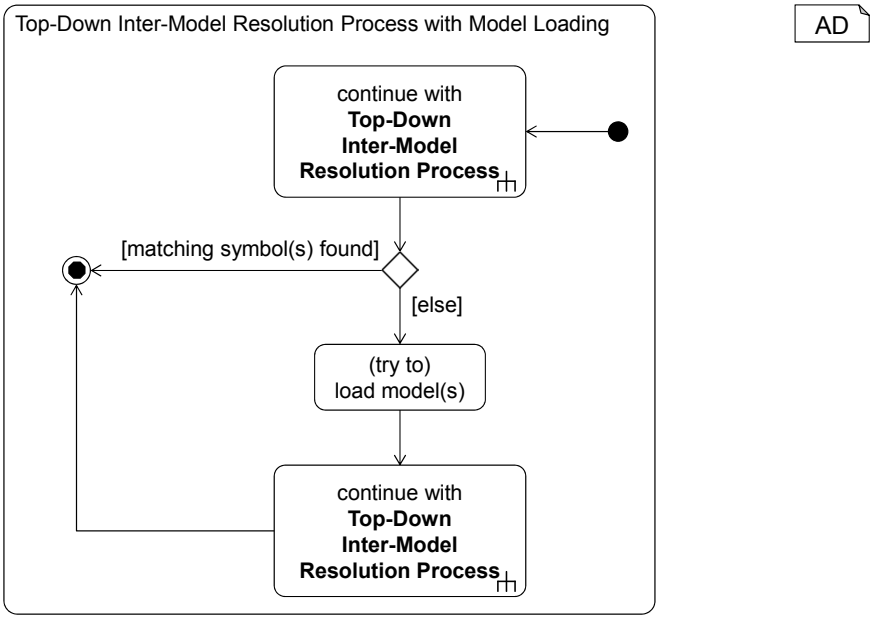


Figure 6.34: General process of the top-down inter-model resolution (cf. Figure 6.11) with model loading.

Finally, another top-down inter-model resolution is conducted and finds the symbol if its model was loaded in the previous step. Else, no corresponding symbol definition exists for the specified reference.

6.9.2 Modeling Language Configuration

Figure 6.35 shows the architecture of the types involved in the model loading process. The `GlobalScope` initiates the model loading. The `ModelPath` class provides access to models stored in files. Besides configuring the resolving filters in the `GlobalScope`, `ResolvingConfiguration` is passed to the `ModelLoader` in order to build up the scope graph. For this, `ModelLoader` obtains the AST from an `AstProvider`. A `ModelingLanguage` provides all information and components required to process models of that specific language. Among others, the `GlobalScope` utilizes the `ModelNameCalculator` and the `ModelLoader` components of a language. Given a (qualified) name, the former calculates possible names of the corresponding model. The latter is responsible for loading models and storing them into the `GlobalScope`. Furthermore, `GlobalScope` can refer to several `ModelingLanguages` in order to enable language aggregation (cf. Section 8.3).

The `ModelingLanguage` interface provides access to language-related functionalities, such as parsing and symbol table creation. Each language subtypes this interface (or its

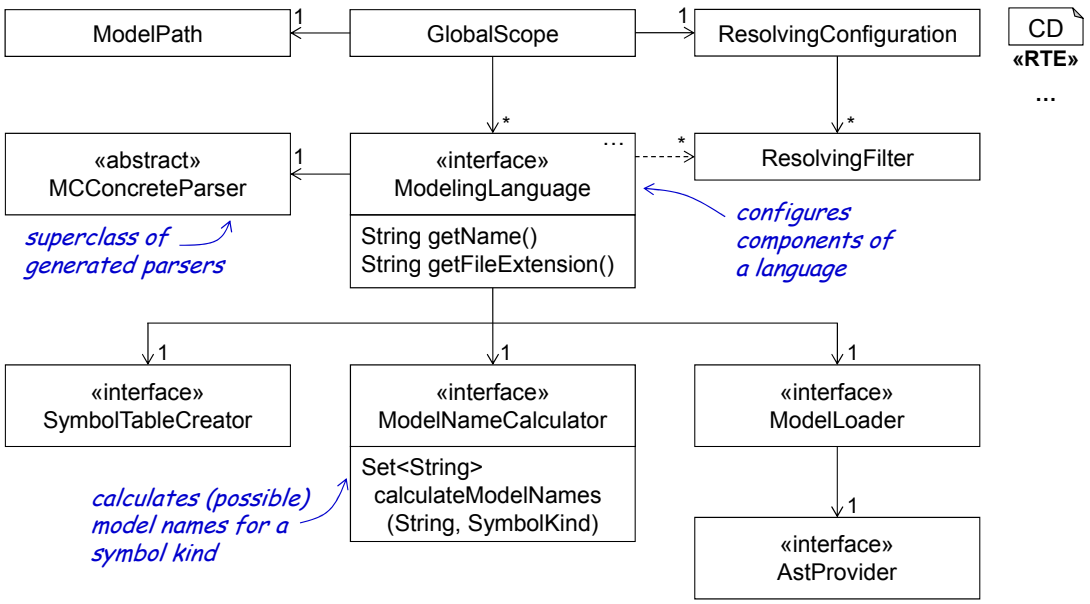


Figure 6.35: Overview of language components provided by the `ModelingLanguage` interface.

default implementation `CommonModelingLanguage`) to enable processing its models. Besides its file extension, a modeling language must provide the (generated) parser for its start rule (cf. Section 2.2.3) and its symbol table creator (cf. Section 5.5). The `ModelLoader` exploits these information to load models of that language (cf. Section 6.9.3).

6.9.3 Model Loader

Figure 6.36 depicts the `ModelLoader` interface provided by SMI which loads models via the name mappings presented above. The method `loadModelsIntoScope`—as implemented in `CommonModelLoader`—tries to load the models with the specified qualified name in the model path. For this, the qualified name is mapped to a corresponding physical file name, including the languages file extension retrieved via `ModelingLanguage`'s `getFileExtension` method. Additionally, `loadModelsIntoScope` creates the scope graph for the found models and adds them to the passed enclosing scope (i.e., the global scope) which is configured with the `ResolvingConfiguration` (cf. Section 5.5). Language-specific model loaders must subclass `CommonModelLoader` and implement its abstract method `createSymbolTableFromAST`. This method is invoked by `loadModelsIntoScope` after the searched model is parsed, i.e., its AST is created which then can be used to build the respective symbol table (via `SymbolTableCreator`).

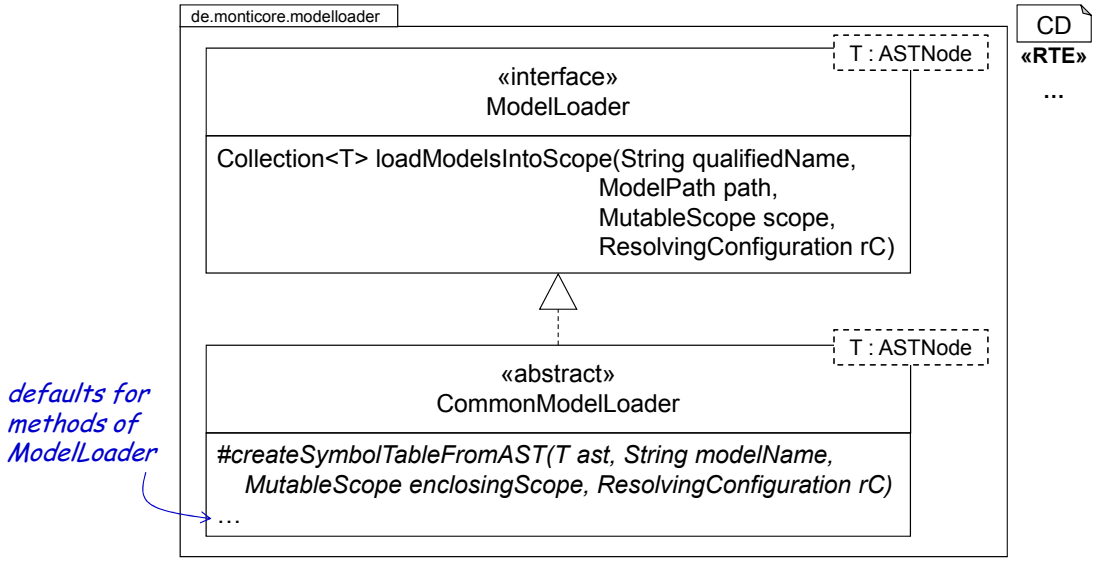


Figure 6.36: `ModelLoader` interface and its default implementation `CommonModelLoader`.

6.9.4 AST Provider

In the examples above, the models are stored in files. However, IDEs such as Eclipse hold the ASTs of the parsed models already in the memory. To enable model loading independent of the physical manifestation, SMI provides the interface `AstProvider` which is utilized by `ModelLoader` to retrieve the AST of a model. By default, `ModelLoader` exploits the `FileBasedAstProvider` to conduct the file-based processing, mentioned above. For usage in Eclipse the `EclipseAstProvider` class can be employed to reuse the AST created during Eclipse's parsing process.

6.9.5 Model Name Calculator

To calculate the possible model names of a symbol as described above, SMI provides the `ModelNameCalculator` and its default implementation `CommonModelNameCalculator`. The method `calculateModelNames` determines a set of possible model names depending on the symbol's name (first parameter) and kind (second parameter). The default implementation in `CommonModelNameCalculator` solely returns the passed symbol name regardless of the kind, and hence, treats each symbol kind as a top-level element. If a language allows for the definition of inner elements, a language-specific `ModelNameCalculator` must be implemented and override the `calculateModelNames` method. Listing 6.37 demonstrates this by the example of Java symbols.

```java
public class JavaModelNameCalculator
        extends CommonModelNameCalculator {

  @Override
  public Set<String> calculateModelNames
                     (String name, SymbolKind kind) {
    if (JavaTypeSymbol.KIND.isKindOf(kind)) {
      return calculateModelNamesForType(name);
    }
    else if (JavaFieldSymbol.KIND.isKindOf(kind)) {
      return calculateModelNamesForField(name);
    }
    else if (JavaMethodSymbol.KIND.isKindOf(kind)) {
      return calculateModelNamesForMethod(name);
    }
    return Collections.emptySet();
  }
  // ...
}
```

Java «LS»

Listing 6.37: Exemplary implementation of a language-specific model name calculator.

`JavaModelNameCalculator` extends `CommonModelNameCalculator` and overrides its method `calculateModelNames` (lines 4–17, Listing 6.37). For each symbol kind, a method exists that calculates the possible model names depending on this kind (lines 8, 11, and 14). That way, the model name of *p.C* is *p/C.java* or *p.java* depending on whether a type kind or a field kind (or method kind) is specified. For other kinds, an empty set is returned (line 16).

6.10 Example of Usage

To conduct all phases of the name resolution, `GlobalScope` must be configured appropriately which includes registration of all involved languages (cf. language aggregation in Section 8.3) as well as specification of the model path(s) (cf. Section 2.2.6). Listing 6.38 shows an example of configuring and using the global scope. The global scope (line 4) is configured with the Java language configuration (line 1, Listing 6.38) and the model path `src/main/models` (lines 2–3). As it can be seen, the `GlobalScope` instance is assigned to `Scope gs` since the concrete scope type is not required for the resolution.

After the configuration, the global scope can be exploited for the (inter-model) resolution. In line 7, the Java type symbol `p.E` is resolved (assuming the case in Figure 6.9 on page 133). If the artifact *p/E.java* is not loaded yet, it will be lazily loaded from the model path `src/main/models` via the model loader registered in `JavaLanguage`.

```java
JavaLanguage javaLanguage = new JavaLanguage();
ModelPath modelPath =
  new ModelPath(Paths.get("src/main/models"));
Scope gs = new GlobalScope(modelPath, javaLanguage);

// resolves to class in p/E.java (in the specified model path)
gs.resolve("p.E", JavaTypeSymbol.KIND);

// resolves to field f of class D defined in p/q/D.java
Optional<JavaFieldSymbol> fieldSymbol =
  gs.resolve("p.q.D.f", JavaFieldSymbol.KIND);

// starts bottom-up intra-model resolution phase
fieldSymbol.get().getEnclosingScope()
  .resolve("T", JavaTypeSymbol.KIND);
```

Java «HC»

Listing 6.38: Exemplary configuration and usage of SMI's resolution mechanism.

Analogously, the second resolution request (lines 10–11) resolves to the field `f` in the class `p.q.D` (assuming the case in Figure 6.10 on page 133). For this, the model name calculator (registered in `JavaLanguage`) determines the possible model names of the requested field since `JavaFieldSymbol.KIND` is specified (line 11). The resolved field symbol is assigned to `fieldSymbol` (line 10). This allows to start the resolution from the field's enclosing scope `D` (lines 14–15). In contrast to the global scope, `D` is defined within an artifact, and hence, its `resolve` method starts the bottom-up intra-model resolution phase.

As demonstrated in this example, after a minimal configuration the resolution process is conducted including name qualification and model loading. The language user does not have to consider the different resolution phases introduced throughout this chapter. Instead, a scope's `resolve` method completely encapsulates the resolution process and proceeds with other phases if required.

6.11 Related Work

Many frameworks provide a generic, language-unspecific resolution mechanism and additionally allow for language-specific customization. They, among others, differ in their default implementations (if any) and whether they provide explicit concepts to realize the language features discussed in this chapter. However, none of the frameworks explicitly provides all four resolution phases as introduced in this chapter. This phases simplify the complex resolution process by dividing it into smaller parts. Furthermore, they facilitate easy and efficient customization of the resolution process by allowing the

language engineer to focus on the respective phases, as elaborated in Sections 6.8 and 6.9. Our resolution process highly depends on the elements in the scope graph, e.g., whether a scope is a shadowing or a visibility scope. Those concepts are already introduced and discussed with related work in Chapters 3 and 4. This section discusses technical details based on these two chapters.

The meta-language NaBL [KKWV13] integrated in the Spoofax language workbench [KV10] enables a declarative specification of name resolution as well as language features such as shadowing, overloading, and imports. In contrast, the current thesis and also Xtext [Bet13], Völkel [Völ11], Parr [Par10], and MPS [VS10] specify these features programmatically, e.g., via Java. Since NaBL's expressiveness is limited to these features, Stratego [BKVV08] can be exploited for more complex name resolution rules.

NaBL models are generated to Stratego rules which employ Spoofax's index API. The API provides lookup methods to retrieve the respective definitions of annotated identifiers (cf. Section 5.1). Query methods allow for obtaining information associated with those definitions. Same as the `resolveLocally` method of the `Scope` interface in the current thesis (cf. Section 4.2.2), the Stratego rule `index-lookup-one` searches for a matching definition within the (direct) enclosing scope (cf. [KKWV13]). `index-lookup` employs `index-lookup-one` for each enclosing scope until reaching the top-level scope. This corresponds to our bottom-up resolution phase conducted via the `resolve` methods depicted in Figure 6.20. These methods, however, standardly continue with the top-down resolution phases which include partially qualified names. This corresponds to NaBL's `index-lookup-all-levels` method. The technical methods for symbol resolution in our approach each returns a symbol (or a collection of symbols) which represents both a definition and its associated information (cf. Section 3.3 and Section 3.8). In contrast, in NaBL the resolved definitions are retrieved via one of the `lookup` rules above and are passed to, for example, `index-get-data` to obtain its associated information stored in the index. NaBL enables (partial) specification of type resolution. That way, expressions such as method invocations can be resolved to the respective method definition, including its correct signature. In the current approach the name resolution can employ the type resolution only *programmatically*. This corresponds to the approaches in, e.g., [EB10, Völ11].

In [Völ11], the `Resolver` class conducts the generic name resolution and employs specific resolver clients (i.e., the `IResolverClient` interface) for the language-specific aspects (cf. discussions in Section 4.2.2 and Section 4.3). A resolver client conducts all resolution phases (if required) presented in the current thesis for a specific symbol table entry kind. In contrast, the approach in SMI only requires customization for the specific phase. Since resolvers in [Völ11] allow for arbitrary specification of the resolution process, they do not ensure that entries in the namespaces behave uniformly (cf. *RRQ3*). For example, if resolvers traverse the namespace hierarchy differently this could lead to different results. Furthermore, since namespaces in [Völ11] are completely generic and solely serve as containers for symbol tables (cf. Section 4.2), they do not embody any

information that is relevant for name resolution. For instance, the concept of named and unnamed namespaces does not exist (cf. Def. 3.9), hence, there is no possibility to generically traverse the namespace hierarchy top-down as discussed in Section 6.5. Furthermore, a global namespace (analogously to `GlobalScope` in SMI) does not exist explicitly in [Völ11]. Hence, the namespace hierarchies of different models are not part of a single namespace graph but are rather connected programmatically. Resolution of inner elements is therefore not possible in a generic way but instead must be implemented for each specific language.

Same as in our approach, the resolver in [Völ11] facilitates resolving via arbitrary additional information. However, as described in Section 4.5 access modifiers are no top-level concepts in [Völ11], and hence, resolution via access modifiers is not provided in a generic way, as in the current thesis. Since no kind hierarchy exists as in our approach (cf. Section 4.1), a dedicated resolver is required for each symbol table entry kind. Similar to our approach, a model loader exists in [Völ11] which, among others, retrieves the AST of the loaded model. However, it does not include the creation of a model's symbol table, as elaborated in Chapter 5. Section 5.1 already mentions that the symbol table creation process in [Völ11] includes qualifying of referenced entries. This is conducted via a dedicated workflow resulting from the sequential processing of models in previous MontiCore versions (cf. [Kra10]). In contrast, in our approach name qualification is conducted dynamically as part of the resolution process. That way, SMI frees the language engineer from explicit handling of the whole name qualification process. Instead, the resolution process is fully reused. The language engineer solely has to specify a `QualifiedNamesCalculator`—if the default implementation does not fit (cf. Section 6.8.2)—which determines possible qualified names depending on import and package information. Given those potential names, the remaining resolution process tries to qualify the name considering the whole name resolution semantics of the language including model loading. In contrast, the infrastructure in [Völ11] requires a so-called `IQualifierClient` which provides two `qualify` methods. The first method qualifies an entry (having the state `UNQUALIFIED`, cf. Section 4.1) starting from its enclosing namespace and *only within the model*. The second `qualify` method qualifies the entry via a `ModelNameQualifier` which extends the search to include models (using a model loader). However, since the qualification is based on artifacts (i.e., files), it does not allow for name qualifying of inner elements in a generic way. Furthermore, loading the full version of an entry also requires a language-specific implementation using the model loader. For this, each symbol table entry implements a specific method.

In summary, one major drawback in Völkel's [Völ11] approach is that name qualification, name resolution, and loading of full versions of symbol table entries are technically separated and only make partial use of each other. This does not only complicate the language development process but also hampers consistency. Additionally, classes such as `STEntry` (cf. Section 4.1) are polluted with technical code. As already discussed in Section 4.3, the resolution in [Völ11] cannot be started with solely a namespace or

a symbol table entry (in contrast to a scope or a symbol in this thesis). Instead, an instance of `Resolver` is required which must be provided by the underlying framework.

Xtext [Bet13] conducts name resolution via its linking and scoping concepts. Linking binds a reference to its respective definition and for this it employs scoping. As already discussed in Section 3.5 and Section 4.2, Xtext's scoping concept differs from the one of the current thesis. In Xtext, a scope is considered from a reference's viewpoint containing a list of (potential) definitions the reference resolves to and is represented by the `IScope` interface. `IScopes` can be nested and standardly are traversed bottom-up and that way, among others, consider shadowing. Xtext provides the `IScopeProvider` interface (which is subtyped for each language) with its method `getScope(EObject, EReference)` which returns an `IScope` for the reference (second parameter) in a specific context, e.g., a method declaration (first parameter). The `getScope` method is very generic and requires a lot of type introspections in order to implement language-specific scopes (cf. [Bet13]). Alternatively, Xtext provides the `AbstractDeclarativeProvider` (and many others) which facilitates specifying scopes in a declarative way using name conventions based on the Ecore model (emerged from the grammar definition)[5]. While this approach allows for efficient development of scopes, it is error-prone when the grammar changes or the language engineer accidentally states wrong names, resulting from the fact that reflection is used. In contrast, scopes and symbol references in the current thesis are typed and incorporate the resolution logic. That way, type introspection is omitted and the resolution is not affected by changes in the grammar (only the symbol table creation might be affected, cf. Chapter 5). Xtext's approach is very flexible since it enables specifying scoping for specific references in specific contexts. In SMI, this can be (partially) realized via concrete `SymbolReference` subtypes (cf. Section 4.4). The `getReferencedSymbol` method can be implemented in an arbitrary way, also considering its context (e.g., the enclosing scope). However, this should be avoided (if possible) since it may violate *RRQ3*. In contrast to our approach, Xtext does not provide explicit concepts for access modifiers. Instead, language-specific implementations must consider this (same as [Völ11]). For elements visible in the global scope, Xtext provides the `IGlobalScopeProvider` (and its default `ImportNamespacesAwareGlobalScopeProvider`).

Similar to `Scope` in SMI, MPS [VS10] provides a `ScopeProvider` interface for so-called *inherited scopes* which must be implemented by elements (i.e., concepts) that define other elements. It provides a `getScope(kind, child)` method which is similar to `Scope`'s `resolve` methods of the current thesis. The `child` parameter is the child node from which the resolution request is started (e.g., a reference). `getScope` returns a `Scope` which is a collection of possible definitions of a reference (same as in Xtext [Bet13]). When ignoring the `child` parameter the returned `Scope` in particular consists

[5] Customization in Xtext is mainly configured via Google's dependency injection framework *Guice* (`https://github.com/google/guice`).

of the containing elements, same as `Scope` in our approach. The default implementation employs a bottom-up resolution including shadowing. Same as Xtext, MPS provides many defaults for scope calculation, such as `ModelPlusImportedScope` which also considers imported models (similar to `resolveImported`). Alternatively, MPS allows for defining scopes directly in the respective reference, i.e., the reference itself calculates which elements it potentially refers to, similar to `SymbolReference` in SMI.

The `Scope` interface in [Par10], among others, provides a `resolve(String)` method which enables resolving a symbol with the specified name, similar to our approach and MPS. As already discussed in Section 4.1, symbol kinds are not distinguished, hence, cannot be specified for the resolution. Parr [Par10] suggests a dedicated method `resolveMember(String)` as `resolveImported(...)` in SMI, to search within, e.g., the class hierarchy instead of the enclosing scope tree.

The meta-compilation tool JastAdd [HM03, EH07] employs parameterized attributes [Hed00, Ekm06]. In particular, the value of a parameterized attribute depends on the combination of its parameter values. Technically, JastAdd realizes those attributes via parameterized accessor methods in the AST classes. This, among others, allows for name resolution, as demonstrated by Hedin [Hed11]. For example, the `State lookup(String)` method returns a state depending on the name specified in the parameter. Name analysis in JastAdd can be realized as an combination of inherited and synthesized attributes as well as equations. A local resolution, as in the current thesis, only depends on the information of the current AST node (and eventually its children). Thus, AST nodes can add the method `State localLookup(String)` which searches for a matching state in the AST node itself and its children. In particular, this embodies both a synthesized and parameterized attribute since its result depends (i) on the passed parameter value as well as (ii) on its directly defined fields and the ones of its children. In our approach, the search in the child nodes can imply a top-down resolution. Hence, `State localLookup(String)` is similar to SMI's `resolveDown` methods. If the information cannot be found in the current node (and its children) the search needs to continue with the parent node which, essentially, is the context of the current node [Hed11]. For this, JastAdd employs inherited attributes. This means, the root node (or parent node) defines a method `State lookup(String)` which traverses its children and searches in them locally, i.e., it calls the children's `localLookup(String)` method described above. Since child nodes inherit this attribute, they can access elements defined in the enclosing scope. This corresponds to our `resolve` method. Please note that although `localLookup` traverses its child nodes, it is not same as our `resolveDown` method. Only the calculation is conducted downwards, however, that way child nodes can inherit the method which is part of their context. In particular, child nodes can access information of the parent node which corresponds to our bottom-up resolution. Ekman [EH06] demonstrates an example of utilizing JastAdd for more complex scope structures (such as in Java). Bürger et al. [BKWA11] employ JastAdd's RAGs to formally specify EMF meta-model semantics, which includes references between meta-classes.

Moreover, JastAdd exploits so-called collection attributes [Boy96, MEH09], i.e., values that are composed of contributions distributed in any node of the AST. One special feature of this is that the contributors populate (declaratively) a collection attribute, not the attribute itself. That way, for example, all outgoing transitions of a state can be populated by the respective transition nodes and not by the state node. This feature is not explicitly supported in the current thesis, but can be realized via AST traversals.

6.12 Naming Conventions

Table 6.39 lists naming conventions for improving the readability of language-specific implementations of `ModelingLanguage`, `ModelLoader`, and `ModelNameCalculator`. In particular, the name should consist of the (short) language name and the component's name, e.g., `JavaModelLoader`.

SMI Component	**Convention**	**Example**
ModelingLanguage	suffix "Language" or suffix "ModelingLanguage"	`JavaLanguage` `JavaModelingLanguage`
ModelLoader	suffix "ModelLoader"	`JavaModelLoader`
ModelNameCalculator	suffix "ModelNameCalculator"	`JavaModelNameCalculator`

Table 6.39: Naming conventions for language-specific implementations of `ModelingLanguage`, `ModelLoader`, and `ModelNameCalculator`.

Chapter 7

Generative Engineering of Language-Specific Symbol Table Infrastructures

The generic infrastructure SMI provides default behavior for concepts occurring in many block structured languages (cf. Chapters 4, 5, and 6). It aims at reducing boilerplate code by providing reasonable defaults and that way increase efficiency and effectiveness of symbol table engineering. Still, when developing language-specific symbol tables, there exists a lot of boilerplate code as well as repetitive work that has to be conducted by hand, which includes, among others:

- Relating symbols to their specific kind (e.g., implementing the `KIND` constant).
- Applying the default implementations introduced in the previous chapters for language-specific components.
- Creating language-specific symbol table components, such as model loaders and symbol table creators, and integrating them into the generic infrastructure.

Those aspects concern *language-specific* information, and therefore, cannot be tackled with *generic* features of GPLs. Thus, to further improve the efficiency of the symbol table development process, we apply a model-driven and generative approach (cf. [Rum12, CE00]) which enables to generate parts of a language-specific infrastructure. Following Kelly and Tolvanen [KT08], the generated code builds on the generic infrastructure (i.e., the domain framework).

7.1 Overview and Primary Requirements

As for the generated parsers and context condition infrastructure (cf. Section 2.2), the grammar—i.e., its AST and symbol table—serves as input model for the generation of a language-specific symbol table infrastructure. The participating artifacts are depicted in Figure 7.1. Given the grammar, the generator produces symbol table components such as the specific model loader, symbols, and symbol kinds.

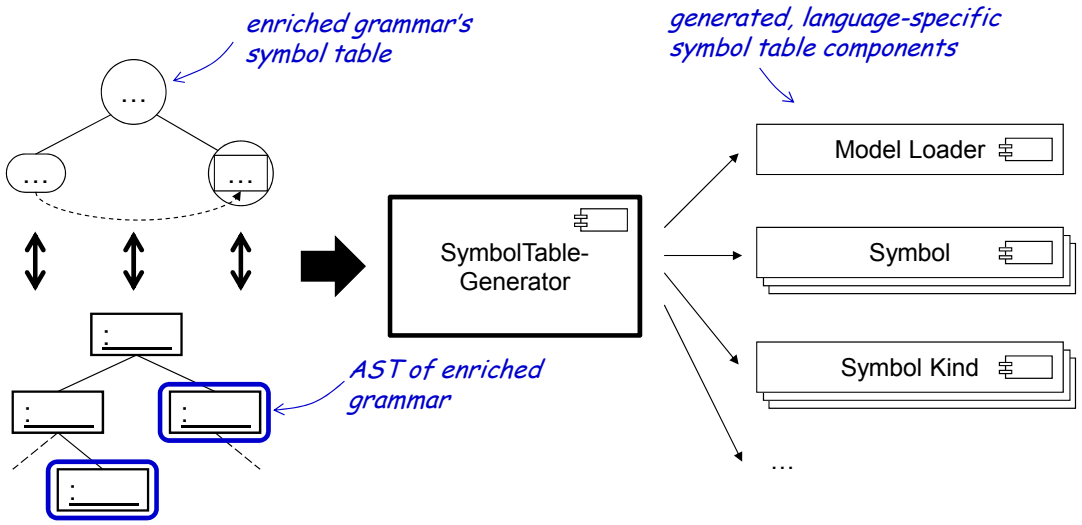


Figure 7.1: Overview of the input and output of the symbol table generator.

In order to improve the symbol table creation, we extended MontiCore's meta-grammar—i.e., the grammar for describing grammars [Kra10]—with further (semantic) information.

The symbol table components, such as symbols and symbol table creators, are highly language-specific. Thus, their full specification would require complex DSL(s) as, for example, used in Spoofax [KV10]. Since language engineering with MontiCore is mainly Java-based, and hence, targets tool developers with skills in Java, the generative approach should not be based on (new) complex DSLs. Instead, the leading requirements are as follows:

GRQ1 (Effectiveness of Generation) The generator should produce *a large part* of the language-specific symbol table infrastructure based on the language's grammar extended with *marginal* additional information.

GRQ2 (Efficiency of Customization) The customization should be conducted with *little effort*, i.e., overhead should be reduced.

GRQ3 (Ease of Learning) The customization of the generated symbol table infrastructure should be specified via *handwritten Java code* instead of a new complex DSL (see above).

GRQ4 (Partial Use of Generated Infrastructure) The language engineer should not be enforced to use the whole generated infrastructure. Instead, she should be able to reuse required parts and ignore the rest.

Driven by the above requirements, we enriched MontiCore's meta-grammar with additional information (cf. Section 7.3) and extended the MontiCore language workbench with a generic handwritten extension mechanism which allows to easily adapt and integrate generated code with handwritten code (cf. Section 7.14).

Benefits

Using the (enriched) grammar with additional information and generating (parts of) the language-specific symbol table infrastructure yields the following benefits (cf. [DK98, MHS05, Rum12, VBD+13]):

- It increases efficiency of the symbol table engineer since redundant information is concentrated at one single place (i.e., the generator) [Rum12].
- It enhances maintainability of the language-specific symbol table components, for example, when the generic infrastructure evolves. This is especially the case when using the proxy pattern for symbol references which must, among others, delegate each method of the `Symbol` interface to the real symbol (pattern *(Q) Symbol Reference Using Proxy Pattern*, cf. Section 4.4.1). Adding new methods to `Symbol` requires *manually* updating all those symbol reference classes. This is aggravated by the fact that missing delegations cannot be determined statically during compilation time, and thus, can lead to silent bugs.

  A generative approach enables to simply adapt the respective generator which entails an automatic update of all generated symbol references to fit into the (updated) infrastructure (cf. Section 7.8).
- It improves the quality since decreasing errors introduced by manually written code and also ensuring that the code adheres to specific (coding) guidelines, for example, those discussed throughout Chapter 4 and naming conventions suggested in Section 4.7 and Section 6.12.
- It further improves readability and understandability of a symbol's references, e.g., that names used in a transition really refer to states (cf. Section 7.2).

Chapter Outline

This chapter introduces the generative approach for developing language-specific symbol tables by the example of a simple automaton language introduced in Section 7.2. Section 7.3 presents MontiCore's grammar format enriched with symbol table specific information. Next, Section 7.4 highlights the architecture of the symbol table generator and also gives an overview of the (partially and fully) generated symbol table components elucidated in the subsequent Sections 7.5 – 7.13. Finally, Section 7.14 illustrates mechanisms for customizing the generated code and integrating handwritten code.

7.2 Automaton Example

An automaton model consists of states and transitions (lines 3–4 in Listing 7.2). A state has a name and can optionally be an initial or a final state (lines 6–8). A transition has a source state, an input, and a target state (lines 10–11). Furthermore, it has a name, which can be employed (from within another model) to specify additional information, such as the transition's preconditions and actions.

The grammar (and hence the resulting AST structure) does not reveal that a transition references two states. It solely employs `Names`. In contrast, the essential model (embodied by the symbol table structure, cf. Def. 3.15) depicted in Figure 7.3 clarifies the relation of transitions and states. The `AutomatonSymbol` represents the whole automaton and provides two methods to retrieve the containing states and transitions from its spanned scope. A dedicated `AutomatonScope` class is not required since the spanned scope behaves exactly as `CommonScope` (cf. Section 4.3).

Unlike `AutomatonSymbol`, `StateSymbol` and `TransitionSymbol` do not span scopes, thus, cannot contain further symbols. Finally, `TransitionSymbol` refers to its states. In this example, we employ the proxy pattern approach (i.e., pattern *(Q) Symbol Reference Using Proxy Pattern*, cf. Section 4.4.1) for the state symbol reference. This pattern is well-suited since a state reference does not hold reference-specific information (such as type arguments). Consequently, `TransitionSymbol` can directly refer to `StateSymbol` (instead of `StateSymbolReference`). Finally, each symbol has its own symbol kind, namely `AutomatonKind`, `StateKind`, and `TransitionKind`.

MCG

```
1 grammar AutomatonDSL_v1 extends Lexicals {
2
3   Automaton =
4     "automaton" Name "{" (State | Transition)* "}";
5
6   State =
7     "state" Name
8     (("<<" ["initial"] ">>" ) | ("<<" ["final"] ">>"))* ";";
9
10  Transition =
11    Name ":" from:Name "-" input:Name ">" to:Name ";";
12 }
```

Listing 7.2: Grammar of the automaton language.

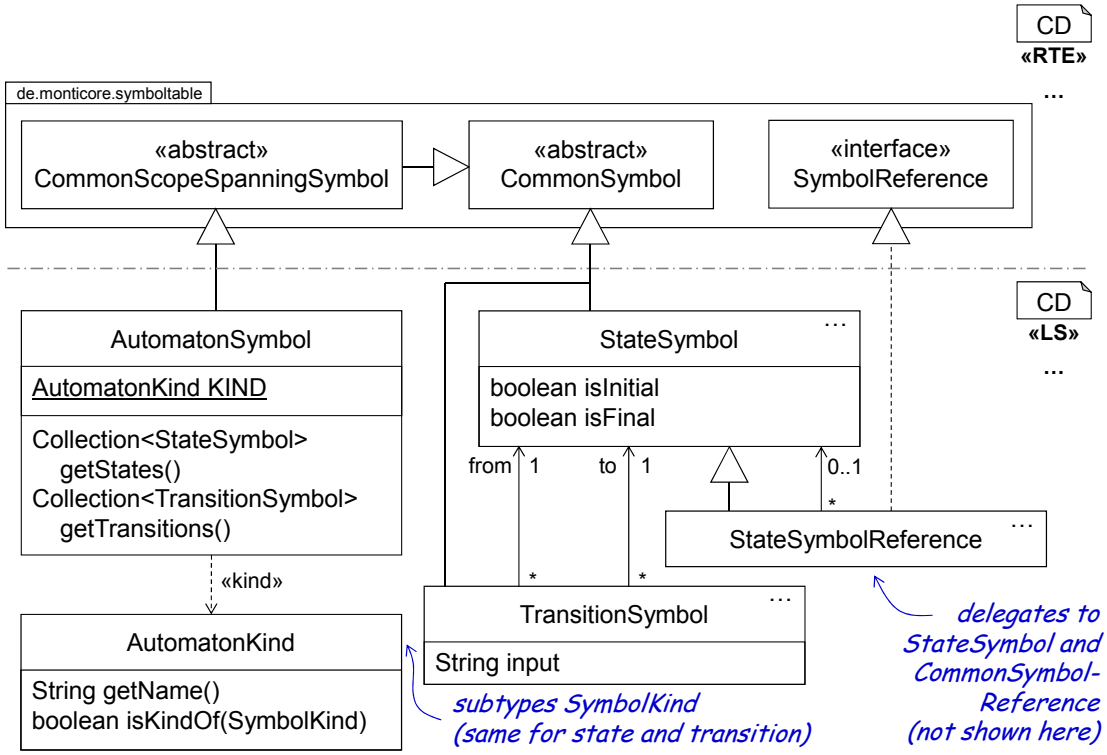


Figure 7.3: Symbol table structure of the automaton language.

7.3 Enriching the MontiCore Grammar with Symbol Table Information

MontiCore's meta-grammar allows for specifying both the concrete syntax and the abstract syntax of a language [KRV07b, KRV10]. From this, a generator derives the corresponding components, such as the parsers and AST classes. The symbol table generator follows this approach and generates a language's symbol table infrastructure (or parts of it) from the grammar.

The grammar typically does not contain all information required for deriving a full-fledged language-specific symbol table from it. For this reason, we first extended MontiCore's meta-grammar with a simple annotation mechanism (cf. [Fow10]) which in particular concerns the production rules, such as `ClassProd`, `InterfaceProd`, and `AbstractProd` as well as the `NonTerminal` rule (cf. Section 2.2).

Listing 7.4 gives an excerpt from the extended grammar and Listing 7.5 shows an instance of it by the example of the automaton grammar. The `InterfaceProd` production (lines 1–4, Listing 7.4) additionally contains the optional nonterminal `SymbolDefini-`

```
1 InterfaceProd implements ParserProd =                                MCG
2   "interface" Name SymbolDefinition?
3   ...
4   ;
5
6 NonTerminal implements RuleComponent =
7   (variableName:Name&? "=" | usageName:Name&?  ":")?
8   Name ReferencedRule?
9   ...
10  ;
11
12 SymbolDefinition = "@!";
13
14 ReferencedRule = "@" Name;
```

Listing 7.4: MontiCore grammar extended with symbol table information.

```
1 grammar AutomatonDSL extends Lexicals {                              MCG
2
3   Automaton@! =
4     "automaton" Name "{" (State | Transition)* "}";
5
6   State@! =
7     "state" Name
8     (("<<" ["initial"] ">>" ) | ("<<" ["final"] ">>"))* ";";
9
10  Transition@! =
11    Name ":" from:Name@State "-" input:Name ">" to:Name@State ";";
12 }
```

Listing 7.5: Grammar for the automaton language enriched with symbol table information.

`tion` (line 2) which allows to mark a production as a symbol definition via `@!` (line 12). In the automaton example in Section 7.2, dedicated symbols exist for the whole automaton, its states, and its transitions. Thus, we mark the corresponding productions with `@!` (lines 3, 6, and 10, Listing 7.5). The annotation does not distinguish between a normal symbol and a scope spanning symbol. Instead, the generator automatically determines it (cf. Sections 7.6 and 7.7).

Furthermore, we extended MontiCore grammar's `NonTerminal` production to optionally contain the name of the rule it (semantically) refers to (line 8, Listing 7.4). For this the new production `ReferencedRule` is introduced (line 14, Listing 7.4). A transition

```
1 grammar AltAutomatonDSL extends Lexicals {                    MCG
2
3   Automaton =
4     "automaton" Name "{"
5       (
6         // state //
7         ("state" stateName:Name
8         (("<<" ["initial"] ">>" ) | ("<<" ["final"] ">>"))*)
9         |
10        // transition //
11        (transName:Name ":"
12         from:Name "-" input:Name ">" to:Name ";")
13      )*
14    "}";
15 }
```

Listing 7.6: Automaton grammar with only one production.

in the automaton language, for instance, refers to two states, defined via `from:Name` and `to:Name` in the grammar. To specify that the names really refer to a state, we annotate them with `@State` (line 11, Listing 7.5). This information is not only useful for the symbol table but also on the grammar level itself. For example, a context condition can check that the referred rule (here `State`) contains a `Name` nonterminal. Moreover, it improves understandability of the grammar since the reference is explicitly stated. Hence, we use the term "referenced rule" instead of "referenced symbol".

The automaton grammar is designed in such a way that it specifies a dedicated production rule for each essential model element [HMSNR15b], namely `Automaton`, `State`, and `Transition`. This not only results in a useful AST structure simplifying its traversal but also facilitates relating AST nodes to symbol table elements (cf. Section 4.6). For instance, the AST node `ASTState` (generated from the production `State`) can be related to the symbol `StateSymbol`.

In contrast, if essential model elements are not specified in a dedicated production less parts of the symbol table can be generated. Listing 7.6 demonstrates an (extreme) example of an automaton grammar where the whole model is described in a single production[1]. Instead of separating states (lines 7–8) and transitions (lines 11–12) into their own productions, they are specified within the `Automaton` production. As a result, only the AST node `ASTAutomaton` is generated which is related to the symbols `AutomatonSymbol`, `StateSymbol`, and `TransitionSymbol`. Please note that the (handwritten) automaton symbol table as depicted in Figure 7.3 still embodies the essence of models of the `AltAutomatonDSL` grammar.

[1] Certainly, this grammar does not follow the guidelines as described, e.g., in [KKP+09].

In summary, grammars like in Listing 7.6 should be avoided since they lead to AST nodes with mixed concerns and also hamper the symbol table generation. Particularly, essential model elements should be specified with dedicated productions.

Alternative Approaches

Tagging languages (e.g., [GLRR15]), delta modeling (e.g., [CHS10, SSA14]), model transformation approaches [MCG05], and aspect-oriented programming [KLM+97] enable to enrich a model with further information without changing its artifact directly. Instead a dedicated artifact is used which yields the advantage that the original model is not polluted. However, introducing new artifacts increases the complexity which impedes both understandability and maintainability of the model (and the additional artifacts).

The grammar extension described above, however, is only marginal, hence, the annotations do not pollute the grammar. Furthermore, annotating nonterminals directly helps to understand the semantics of a language, e.g., that a specific name refers to a state.

Alternatively, a convention-over-configuration approach can be applied in order to determine symbols. In Def. 3.1 we, among others, state that a symbol represents a *named* model element. Following from this, we can produce a symbol for each production containing the `Name` token (defined in the `Lexicals` grammar Section 2.2.2). While `Automaton` and `State` in Listing 7.2 solely contain `Name`, the `Transition` production additionally has different usages of it, e.g., `from:Name` and `to:Name`. If transition contained only these two usages, this could be an indication that a transition does not have a name itself but only employs names of its referenced states. Therefore, a transition would not necessarily need to be represented by a symbol (cf. Section 3.11). To exclude such cases, we could consider only productions containing `Name` either without usage name (i.e., `Name`) or with the usage name `name` (i.e., `name:Name`).

A drawback of this approach is that it can lead to many inappropriate symbols. For example, the production `QualifiedName = Name (|| "." )` fulfills the above criteria, but obviously is not an *essential* model element (cf. Section 3.3). Moreover, the approach's restrictiveness might be a drawback as well; the language engineer cannot specify further symbols for productions not containing the `Name` token (even if they employ another token for defining a name).

Similar to our approach, the language workbench Xtext [EB10] uses the `ID` terminal to set the `name` feature of an element, i.e., `name=ID`. Analogously to `from:Name@State`, Xtext allows to specify the type of the referenced element via `from=[State]` (which is a default for `from=[State|ID]`). From this specification in the grammar, Xtext then generates the abstract syntax based on the Ecore meta-model [SBPM09].

Projectional approaches as applied in, for example, MPS [VS10] focus on the abstract syntax and specify references directly on that structure. Consequently, no further enrichments are required.

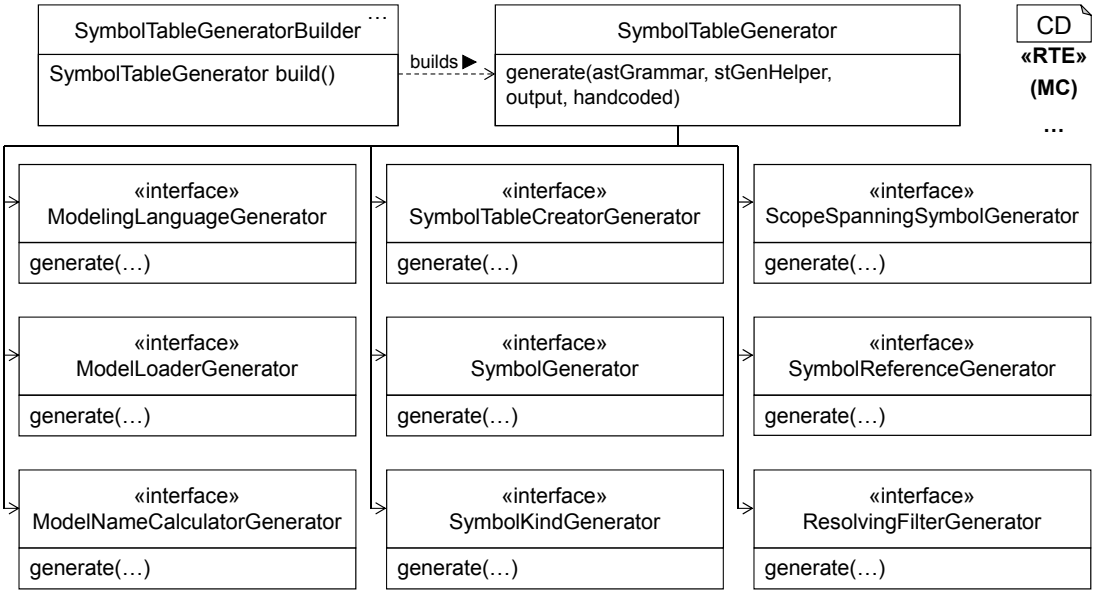


Figure 7.7: Architecture of the symbol table generator. A dedicated generator exists for each symbol table component introduced throughout Chapters 4, 5, and 6.

7.4 Architecture of the Symbol Table Generator

Figure 7.7 presents the architecture of the symbol table generator. As it can be seen, for each component of the symbol table infrastructure introduced in Chapters 4, 5, and 6 a specific generator exists, e.g., `ModelLoaderGenerator` and `SymbolGenerator`. Additionally, a common implementation class exists for each of these generators (not shown in Figure 7.7), e.g., `CommonModelLoaderGenerator`.

`SymbolTableGenerator` is the main class and conducts the generation process via its `generate` method by delegating to the specific generators. For this it requires the following arguments:

ASTGrammar astGrammar The AST of the enriched MontiCore grammar.

SymbolTableGeneratorHelper stGenHelper A helper class which provides useful information for the generation process. Among others, it ensures that only target language conformable names are specified. For instance, given the name `final`—which is a Java keyword—the helper class changes it to `r__final`[2].

File output The output path for generated code (cf. Section 2.2.6).

[2]The helper uses an unnatural prefix such as `"r__"` to prevent clashes with human chosen names (within the same class).

IterablePath handcoded The path of the handcoded classes (cf. Section 2.2.6), which is used to conduct the integration of handwritten and generated code (cf. Section 7.14).

Similarly, each specific generator provides a `generate` method which requires besides the `SymbolTableGeneratorHelper` and `IterablePath` stated above, the `GeneratorEngine` (cf. Section 2.2.1) which ultimately conducts the generation and is configured with the output path of the generated code. Additionally, each generator needs specific information which is described in the following sections. The generators are based on the template-based (cf. [CH03, CH06, Fow10]) approach introduced in [Sch12] and can be customized as shown there. Finally, `SymbolTableGeneratorBuilder` builds [GHJV95] an instance of `SymbolTableGenerator` using the above mentioned default implementations (unless otherwise stated).

Overview of Generated Symbol Table Components

Although a language's symbol table (including its essential model) comprises language-specific aspects, a large part of it still can be generated. Table 7.8 gives an overview of the generated symbol table components.

As it can be seen, the generator produces each component at least partially (cf. *GRQ1*). Symbol kinds, model loaders, and resolving filters can even be fully generated when adhering to the default implementations discussed in previous chapters. Occasionally, the other symbol table components can be fully generated as well (marked with *(x)*). This, above all, depends on whether the grammar covers all information required for specifying that component. Certainly, this implies that AST and symbol table do not differ much.

The subsequent sections particularize for each symbol table component (i) what its typical boilerplate code is and (ii) which parts of it can be generated (iii) based on which information. Two kinds of boilerplate code can be distinguished. First, code that is required to integrate the symbol table components with the generic infrastructure

ST Component	**Partially**	**Fully**
Symbol Kind		x
Symbol	x	(x)
Scope Spanning Symbol	x	(x)
Symbol Reference	x	(x)
Symbol Table Creator	x	(x)
Model Name Calculator	x	(x)
Model Loader		x
Resolving Filter		x
Modeling Language Configuration	x	

Table 7.8: Partially and fully generated symbol table components.

SMI. For example, each scope must subclass `CommonScope`. The respective code (i.e., `extends CommonScope`) is independent from the specific language, and thus, does not require grammar information. Generating it can serve as starting point and documentation for the language engineer. The second kind of boilerplate code is language-specific, e.g., the class name of each symbol table component depends on information which can be derived from the grammar (cf. naming conventions in Sections 4.7 and 6.12).

7.5 Generation of Symbol Kinds

The generator produces a dedicated symbol kind class for each symbol, following pattern *(G) Different Symbol Kinds for Similar Model Elements* (cf. Section 4.1.3). Furthermore, it focuses on pattern *(H) Separating a Symbol and Its Kind into Different Classes* (cf. Section 4.1.4), where a symbol kind does not contain information specific to a symbol (and thus can be fully generated).

Boilerplate Code

With regard to the applied patterns mentioned above, each language-specific symbol kind must (cf. Section 4.1):

1. subtype the `SymbolKind` interface (i.e., integration with SMI),
2. implement `getName` to return the kind's unique name, and
3. implement `isKindOf` as shown in Listing 4.4 (on page 55).

The other two implementation patterns discussed in Section 4.1.4, i.e., *(I) Symbol Class Implements Kind Interface* and *(J) Same Class For a Symbol and Its Kind* require symbol specific information, listed in Section 7.6. Moreover, the last pattern results in one class (instead of two) representing both the symbol and its kind which hampers customization (cf. Section 7.14).

Generative Approach

Integrating the generated class with SMI (see 1.) does not require language-specific information. Similarly, the `isKindOf` method (see 3.) does not require language-specific information. The remaining point solely requires the symbol kind's name, e.g., **`State`**`Kind.class.getName()` which equals the annotated production's name (i.e., `State@! = ...`).

In general, we generate for each production `Prod` marked with `@!` (cf. Section 7.3) a symbol kind class *`Prod`*`Kind` (cf. [HMSNR15b]). Considering the automaton grammar, these are `AutomatonKind`, `StateKind`, and `TransitionKind` (resulting from the nonterminals `Automaton`, `State`, and `Transition`, respectively).

7.6 Generation of Symbols

For the generation of symbol classes the generator applies patterns *(E) Different Symbol Classes for Similar Model Elements* (cf. Section 4.1.2) and *(H) Separating a Symbol and Its Kind into Different Classes* (cf. Section 4.1.4). That is, the generator produces a symbol for each (essential) model element separated from its respective symbol kind class (cf. Section 7.5).

Since a symbol is highly language-specific and depends on several design decisions of the language engineer (cf. Section 3.11), it typically cannot be fully generated.

Boilerplate Code

Although symbols are highly language-specific, they still bear some boilerplate code (cf. Figure 7.9). That is, they:

1. extend the `CommonSymbol` class (i.e., integration with SMI),
2. provide a respective constructor which internally sets the correct kind, and
3. define a public constant `KIND` containing an instance of the corresponding symbol kind.

Generative Approach

The generator needs the following information to generate some language-specific aspects:

- the production name as the symbol's class name (e.g., **`State`**`Symbol`) as well as its constructor (see 2.),
- the production name as type for the symbol's kind (e.g., **`State`**`Kind KIND`, see 3.),
- nonterminals annotated with `@` (e.g., `from:Name@State`) for setting the associations, for example, the `from` and `to` associations of `TransitionSymbol`, and
- semantically relevant terminals (e.g., `["initial"]`, cf. Section 2.2.2) as the symbol's attributes, for example, the **`initial`** field of `StateSymbol` and the respective getter and setter methods `is`**`Initial`** and `set`**`Initial`**.

Additionally, the generator must determine whether to produce a normal symbol or a scope spanning symbol. Considering the automaton grammar in Listing 7.5 on page 172, the `Automaton` production (lines 3–4) contains nonterminals for `State` and `Transition`. Since the corresponding `State` and `Transition` productions both are annotated with `@!`, it follows that an `AutomatonSymbol` contains `StateSymbols` and

TransitionSymbols. Hence, AutomatonSymbol is a scope spanning symbol. The generation of scope spanning symbols is described in Section 7.7.

In contrast to Automaton, neither the State production nor the Transition production contains a nonterminal whose production is annotated with @!. Consequently, StateSymbol and TransitionSymbol are symbols that do not span a scope.

While generating symbol classes reduces manually written boilerplate code, it is not always optimal to generate these with all information that can be derived from the grammar. This is because undesired information in a generated class might hamper its reuse since Java (i.e., the target code) does not allow for removing methods and fields of a class in a non-invasive way. As an example, while subclasses can override methods inherited from the superclass, they cannot remove inherited methods.

For this reason, the symbol table generator always produces two classes per symbol using a naming convention. For each production Prod annotated with @! it generates a *Prod*SymbolEMPTY as well as a class *Prod*Symbol [HMSNR15b]. The first class does not contain any specific attributes. Its main concern is to fit into the generic infrastructure, i.e., extend CommonSymbol and provide a KIND. It allows to easily apply the first implementation pattern presented in Section 4.1.1, that is, the symbol does not contain any information already contained in the AST node.

The second generated class subclasses *Prod*SymbolEMPTY and additionally contains further information directly derived from the grammar (similar to the AST). Finally, the language engineer can decide—depending on her requirements—which of these two classes (if any) to reuse.

The StateSymbolEMPTY class in Figure 7.9, for instance, fits into the generic infrastructure (e.g., subclasses CommonSymbol) and contains (generated) boilerplate code, such as the constant KIND. It does not contain any further information concerning a state. For this purpose, class StateSymbol is generated, which subclasses StateSymbol-EMPTY and extends it with additional information derived from the State production. Now, if StateSymbol contains undesired information, the language engineer can ignore it and instead develop a handwritten class that extends StateSymbolEMPTY and optionally add further language-specific information. That way, the generator transposes most of the boilerplate code mentioned above. MontiCore ensures that the generated symbol class does not interfere with the handwritten class (cf. Section 7.14) (*GRQ4*).

In summary, generating two classes for a symbol simplifies reuse for the language engineer in cases the generated classes do not (completely) fit the requirements. If, for instance, the generated *Prod*Symbol class contains some unneeded information, *Prod*SymbolEMPTY can still be reused. This, above all, reduces the manually written boilerplate code (i.e., subclassing CommonSymbol, adding constant KIND, etc.). It also serves as starting point for the language engineer and demonstrates how to integrate language-specific symbols into the generic infrastructure. Section 7.14 presents the different adapting mechanisms for the generated symbol table infrastructure.

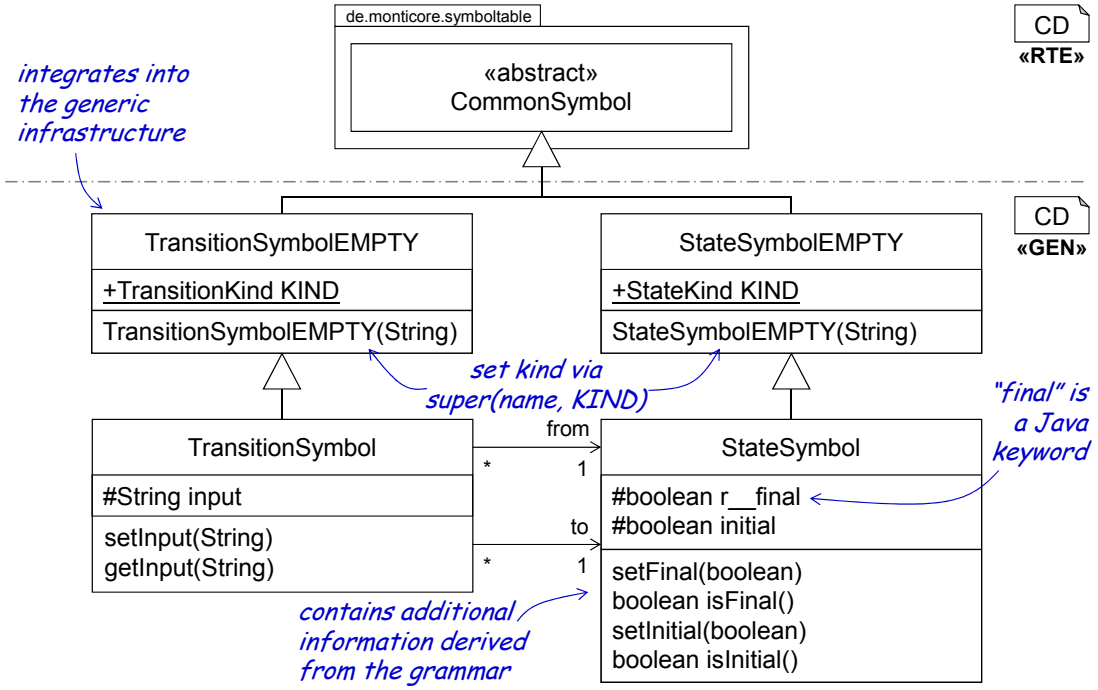


Figure 7.9: Generated symbol classes using the example of transitions and states.

7.7 Generation of Scope Spanning Symbols

For the generation of scope spanning symbols, we apply pattern *(K) Separating Symbol and Its Spanned Scope into Different Classes* (cf. Section 4.3.1). Consequently, this pattern enables adapting the generated symbol and scope classes independently from each other. In contrast, pattern *(L) Same Class for Symbol and Its Spanned Scope* (cf. Section 4.3.1) uses one class for both, and thus, hampers customization. Nevertheless, both patterns require the same information to be generated and produce similar boilerplate code. Same as for normal symbols, scope spanning symbols are only partially generated (in most cases). The remainder of this section focuses on the (preferred) pattern *(K) Separating Symbol and Its Spanned Scope into Different Classes* (cf. Section 4.3.1).

Boilerplate Code

Implementing scope spanning symbols includes the same boilerplate code as for symbols (cf. Section 7.6) with the difference that `CommonScopeSpanningSymbol` (cf. Section 4.3) is subclassed (instead of `CommonSymbol`). Moreover, each scope spanning symbol must:

1. override the (factory) method `createSpannedScope` of `CommonScopeSpanningSymbol` if the symbol spans a specific scope (cf. Figure 4.22), and
2. provide methods for retrieving locally defined symbols from the spanned scope.

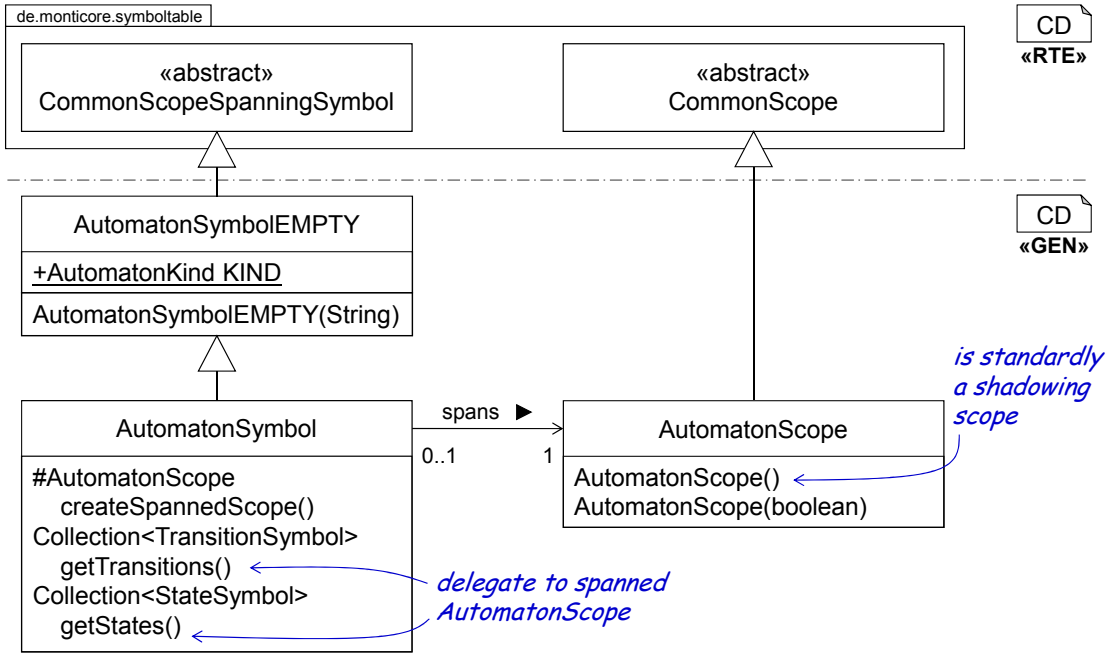


Figure 7.10: Generation of scope spanning symbols using the example of automatons.

Generative Approach

As already mentioned in the previous section, the annotated `Automaton` production entails the generation of the scope spanning symbol `AutomatonSymbol`. In general, an annotated production `Prod` leads to the generation of a scope spanning symbol if it contains at least one nonterminal that belongs to an *annotated* production `Prod2` (where `Prod2` can be `Prod` itself). On the symbol level this means that `ProdSymbol` contains `Prod2Symbol`, and hence, is a scope spanning symbol. The `Automaton` production, for example, is annotated and contains a nonterminal of the annotated production `State`. Hence, `AutomatonSymbol`—generated from the `Automaton` production—is a scope spanning symbol (cf. Figure 7.10) since it contains `StateSymbols`.

Generating scope spanning symbols requires:

- all information as for normal symbols, listed in Section 7.6,
- the annotated production's name to generate a class for the spanned scope, e.g., **`Automaton`**`Scope` and instantiating it in the `createSpannedScope` method of the spanning symbol (see 1.), and
- names of all annotated productions whose nonterminals are used in order to generate those methods stated in the second item.

Java «GEN»

```
1 public class AutomatonSymbol
2                     extends AutomatonSymbolEMPTY {
3
4   public AutomatonSymbol(String name) {
5     super(name);
6   }
7
8   @Override
9   protected AutomatonScope createSpannedScope() {
10    return new AutomatonScope();
11  }
12
13  public Collection<TransitionSymbol> getTransitions() {
14    return spannedScope.resolveLocally(TransitionSymbol.KIND);
15  }
16
17  public Collection<StateSymbol> getStates() {
18    return spannedScope.resolveLocally(StateSymbol.KIND);
19  }
20 }
```

Listing 7.11: Generated `AutomatonSymbol` class.

As shown in Figure 7.10, same as for `State` and `Transition`, two classes are derived from the annotated production `Automaton`, namely `AutomatonSymbolEMPTY` and its subclass `AutomatonSymbol`. Unlike the case in Section 7.6, the former extends `CommonScopeSpanningSymbol`.

Additionally, `AutomatonScope` is generated, which is the spanned scope of `AutomatonSymbol`. Therefore, `AutomatonSymbol` overrides the method `createSpannedScope` of `CommonScopeSpanningSymbol` and returns an `AutomatonScope` (lines 9–11 in Listing 7.11).

Since a symbol serves as a (language-specific) view of its (generic) spanned scope (cf. Section 4.3), the methods `getTransitions` and `getStates` delegate to `AutomatonScope` in order to retrieve the containing `TransitionSymbols` and `StateSymbols`, respectively (lines 13–19 in Listing 7.11). That way, the generator ensures that the guidelines introduced in Section 4.3 are applied.

`AutomatonScope` behaves exactly as `CommonScope`, and thus, could be omitted (cf. Section 7.2). However, it still is generated to simplify customization (if needed) via the approaches presented in Section 7.14.

7.8 Generation of Symbol References

Section 4.4 presents three patterns for symbol references whereas the delegation approach (pattern *(P) Symbol Reference Using Delegation*) and the proxy approach (pattern *(Q) Symbol Reference Using Proxy Pattern*) are preferred. The latter leads to much more boilerplate code since it delegates each method from the proxy to the real symbol. Hence, handcrafting it is more tedious and error-prone than the first approach. For this reason, the generator produces symbol reference classes following the proxy pattern approach to simplify the work of the language engineer. In cases the engineer prefers the delegation pattern, she can employ the customization approaches presented in Section 7.14.

Boilerplate Code

Implementing a reference via the proxy pattern produces a lot of boilerplate code since it:

1. implements the `SymbolReference` interface (cf. Section 4.4),
2. contains an association to `CommonSymbolReference` (which ultimately links to the symbol definition),
3. implements all methods of `SymbolReference` and delegates them to `CommonSymbolReference`,
4. overrides all methods inherited from `CommonSymbol`—and `CommonScopeSpanningSymbol` in case the referenced symbol spans a scope—and delegates them to the referenced symbol,
5. extends the class of the real symbol (i.e., the referenced symbol), and
6. overrides all methods specific to the real symbol's class and delegates them to the referenced symbol.

Generative Approach

The first four items concern only aspects regarding the generic infrastructure, hence, they do not rely on any language-specific information. Therefore, the respective code can be generated independently from the language's grammar.

In contrast, the last two items require

- the name of the annotated production for the real symbol to set the name of the reference class (e.g., **`State`**`SymbolReference`) as well as extending the real symbol's class (i.e., `extends` **`State`**`Symbol`) (see 5.), and
- information about the terminals and nonterminals as described in Section 7.6 in order to generate the methods mentioned in item 6.

Please note that generating symbol references following pattern *(R) Same Class for a Symbol Definition and Its References* (cf. Section 4.4.1) requires the same information as above. In contrast, the delegation approach presented in pattern *(P) Symbol Reference Using Delegation* (cf. Section 4.4.1) solely needs the annotated production's name for setting the reference class' name.

If a handcoded class for the real symbol exists (cf. Section 7.6), the generator cannot rely on the corresponding production anymore when producing methods derived from it (see 6.). For example, when omitting the `isFinal` method in the handcoded `StateSymbol` (which extends `StateSymbolEMPTY`), the `StateSymbolReference` class may not delegate to that method since it would introduce compilation errors.

As a consequence, methods stated in item 6. are omitted (or at least commented out) in the generated `StateSymbolReference` class if `StateSymbol` is handwritten. Alternatively, the generator can make use of Java's parser and symbol table (cf. [Mul15]) in order to process the handwritten classes and obtain their methods. With this information the reference class can be generated more precisely (cf. [MSNRR16]).

The generator produces for each symbol a corresponding symbol reference class, regardless of whether references of that symbol can exist or not from within the language. In the automaton language, for example, transitions refer to states. No other references are possible, i.e., neither transitions nor automatons can be referred to. Still, the generator produces the classes `TransitionSymbolReference` and `AutomatonSymbolReference` in case these references are required when composing the automaton language with other languages. If not needed, they can simply be ignored (cf. Section 7.14).

As stated in Section 7.1, the generative approach improves maintainability for symbol references applying the proxy pattern (pattern *(Q) Symbol Reference Using Proxy Pattern*, cf. Section 4.4.1). It takes over all the boilerplate code that comprises overriding of `Symbol`'s (or `ScopeSpanningSymbol`'s) methods as well as `SymbolReference`'s methods. Hence, in case these interfaces evolve, solely the symbol reference generator must be adapted.

7.9 Generation of Symbol Table Creators

Chapter 5 elaborates the creation of a model's symbol table (i.e., the scope graph and its contained symbols) in detail and also gives methods for realizing language-specific symbol table creators. The symbol table creator generator follows these methods for producing default implementations and provides options for efficient customization.

Boilerplate Code

A symbol table creator:

1. subtypes the generated language visitor interface (cf. Section 2.2.4),

2. extends `CommonSymbolTableCreator` to obtain the default implementations,
3. provides a respective constructor, and
4. implements the `createFromAST` method with the language-specific root AST node type (cf. Section 2.2) as its parameter.

Furthermore, each case (especially **(b)-(d)**) presented in Section 5.2 comes with its own boilerplate code for which the generator generates respective defaults.

Generative Approach

In order to generate the boilerplate code listed above, the generator mainly requires the grammar's name (e.g., `AutomatonDSL`):

- to set the symbol table creator's class name (e.g., **`AutomatonDSL`**`SymbolTable-Creator`) as well as its constructor (see 3.),
- for subtyping the language's generated visitor (e.g., **`AutomatonDSL`**`Visitor`) (see 1.), and
- for implementing the `createFromAST` method with the correct parameter type (e.g., `AST`**`AutomatonDSL`**`Node`) (see 4.).

Item 2. does not require language-specific information. In summary, based on the grammar's name the generator can produce the general structure of a language-specific symbol table creator and integrate it with SMI.

According to the method depicted in Figure 5.7 (on page 109), AST nodes of the cases **(b)**, **(c)**, or **(d)** require a dedicated implementation of the `visit` method in the symbol table creator class. Moreover, the cases **(b)** and **(d)** additionally require an `endVisit` method. The generator is aware of the presented method and produces the respective Java methods. However, it first has to determine (automatically) to which case a specific AST node belongs[3]. For this, the generator makes use of the enriched grammar as follows. First, AST nodes of productions that are not annotated themselves but contain (at least) one nonterminal of an annotated production, are categorized as case **(b)**. That means, the AST node (or the respective model element) spans a scope. As an example, assume a (simplified) production for an if-block like `IfBlock = "{" Variable* "}";` where the production `Variable` is annotated with `@!`. `IfBlock` itself is not represented by a symbol (since not annotated) but defines variables which are represented by symbols.

[3]Please note that the three phases for finding candidates for symbol table elements are (implicitly) applied here (cf. Figure 3.8 on page 50). The first phase (i.e., *"determine symbols"*) is already realized via the annotated productions. Based on those annotations, the generator can conduct the other two phases (i.e., *"determine scope spanning symbols"* and *"determine scopes"*).

Java «GEN»

```
1 @Override
2 public void visit(ASTIfBlock ast) {
3   //--- activity "create scope" ---//
4   MutableScope scope = create_IfBlock(ast);
5
6   //--- activity "initialize scope" ---//
7   initialize_IfBlock(scope, ast);
8
9   //--- activity "set enclosing-sub scope relation" ---//
10  putOnStack(scope);
11
12  //--- activity "link scope and node" ---//
13  setLinkBetweenSpannedScopeAndNode(scope, ast);
14 }
15
16 protected MutableScope create_IfBlock(ASTIfBlock ast) {
17  // creates visibility scope
18  return new CommonScope(false);
19 }
20
21 protected void initialize_IfBlock
22       (MutableScope scope, ASTIfBlock ast) {
23  // e.g., scope.setName(ast.getName())
24 }
25
26 @Override
27 public void endVisit(ASTIfBlock ast) {
28  removeCurrentScope();
29 }
```

Listing 7.12: Generated `visit` and `endVisit` methods of the symbol table creator for model elements that span a scope (following the method depicted in Figure 5.2 on page 103).

Following the method for a model element that spans a scope (cf. Figure 5.2 and Figure 5.7), the generator produces the corresponding `visit` and `endVisit` methods, as shown in Listing 7.12. For this, the generator produces a template method [WBJ90, GHJV95] (with specific hook methods [Pre95a]) which ensures the right order of the activities suggested by the method for case **(b)** (cf. Figure 5.2). Furthermore, it produces two methods `create_`*`Prod`* (i.e., a factory method [GHJV95]) and `initialize_`*`Prod`* with a default implementation following the first two steps in Figure 5.2, i.e., creating the scope and initializing it.

Since an if-block is *unnamed* (cf. Def. 3.9), a visibility scope (cf. Def. 3.8) is instantiated (line 18, Listing 7.12) that does not require further initialization (lines 21–24). In contrast, if the block is named—technically that means the production contains the terminal `Name` (e.g., `NamedBlock = Name "{" ... "}"`)—a shadowing scope will be created instead and is initialized with the respective name (line 23).

The language engineer can customize the creation and initialization of the scope by simply overriding the methods `create_`*`IfBlock`* and `initialize_`*`IfBlock`*. Typically, the last two steps in the `visit` method remain unchanged, i.e., putting the scope onto the stack and linking it with the AST node.

Finally, the respective `endVisit` removes the scope from the stack (line 28) and ensures a correct stack-based approach (cf. Section 5.3.1).

Listing 7.13 presents the `visit` method generated for AST nodes of case **(c)** using the example of `ASTTransition`. Similar to the previous case, the Template Method pattern approach ensures that the implementation adheres to the process outlined in Figure 5.3 (on page 104). Another advantage of this approach is that it simplifies language inheritance (cf. Section 8.4). That is, the inheriting language can introduce a subclass of `TransitionSymbol` which requires further initialization. Consequently, `create_Transition` and `initialize_Transition` must be overridden to return the subclass and conduct the additional initialization, respectively. For this, the latter can be completely reused, i.e., `super.initialize_Transition(transition, ast)`.

Please note that `initialize_Transition` also sets the state references which only requires the annotated name, e.g., `from@`**`State`**. What remains is only boilerplate code, and thus, can be fully generated (lines 23–29, Listing 7.13). This case does not require an `endVisit` method since no scope is spanned (cf. Figure 5.7).

The generator omits concrete implementations of the `create_`*`Prod`* and `initialize_`*`Prod`* methods if the respective *`Prod`*`Symbol` class is handwritten for the reasons discussed in Section 7.8.

Finally, the last case **(d)** is a combination of the previous two cases (cf. Figure 5.4 on page 105). In addition, the production `Automaton` is not only represented by a symbol but also is the start production of the grammar. Consequently, `ASTAutomaton` is the root node of an automaton model. Following Figure 5.7 and Section 5.4, phase **P4.2** of the symbol table creation—i.e., language-unspecific linking of AST nodes and symbol table elements—should be conducted when the symbol table creation of a model finished, i.e., the root node has been processed. For this, the generator produces the `endVisit` method as shown in Listing 7.14 and starts phase **P4.2** via `setEnclosingScopeOfNodes` in line 5 (cf. Section 5.5).

Java «GEN»

```
1 @Override
2 public void visit(ASTTransition ast) {
3   //--- activity "create symbol" ---//
4   TransitionSymbol transition = create_Transition(ast);
5
6   //--- activity "initialize symbol" ---//
7   initialize_Transition(transition, ast);
8
9   //--- activity "add symbol to enclosing scope"  ---//
10  addToScope(transition);
11
12  //--- activity "link symbol and node"
13  setLinkBetweenSymbolAndNode(transition, ast);
14 }
15
16 protected TransitionSymbol create_Transition(ASTTransition ast) {
17   return new TransitionSymbol(ast.getName());
18 }
19
20 protected void initialize_Transition
21         (TransitionSymbol transition, ASTTransition ast) {
22
23   StateSymbolReference from =
24     new StateSymbolReference(ast.getFrom(), currentScope().get());
25   transition.setFrom(from);
26
27   StateSymbolReference to =
28     new StateSymbolReference(ast.getTo(), currentScope().get());
29   transition.setTo(to);
30 }
```

Listing 7.13: Generated `visit` method for model elements that are represented by a symbol (following the method depicted in Figure 5.3 on page 104).

Java «GEN»

```
1 @Override
2 public void endVisit(ASTAutomaton node) {
3   removeCurrentScope();
4   //--- conduct linking phase P4.2 ---//
5   setEnclosingScopeOfNodes(node);
6 }
```

Listing 7.14: Generated `endVisit` method for model elements that are represented by a scope spanning symbol.

7.10 Generation of Model Name Calculators

The model name calculator determines possible model names from a symbol's qualified name depending on the symbol's kind (cf. Section 6.9.5).

Boilerplate Code

A language-specific model name calculator (cf. Listing 7.15):

1. extends `CommonModelNameCalculator` (cf. Section 6.9.5), and
2. overrides `calculateModelNames` in order to handle each symbol kind of the language.

Java «GEN»

```
1 public class AutomatonDSLModelNameCalculator
2          extends CommonModelNameCalculator {
3
4   @Override
5   public Set<String> calculateModelNames
6                      (String name, SymbolKind kind) {
7     if (AutomatonSymbol.KIND.isKindOf(kind)) {
8       return calculateModelNamesForAutomaton(name);
9     }
10    else if (StateSymbol.KIND.isKindOf(kind)) {
11      return calculateModelNamesForState(name);
12    }
13    else if (TransitionSymbol.KIND.isKindOf(kind)) {
14      return calculateModelNamesForTransition(name);
15    }
16
17    return Collections.emptySet();
18  }
19
20  protected Set<String>
21    calculateModelNamesForAutomaton(String name) { ... }
22
23  // same for states and transitions...
24 }
```

Listing 7.15: Generated model name calculator.

Generative Approach

In order to generate language-specific model name calculators, the following information is required:

- the grammar's name as the generated class' name, e.g., **AutomatonDSL**ModelNameCalculator, and
- the name of all symbols (i.e., annotated productions) to generate specific methods as calculateModelNamesFor**Automaton**(String) (lines 20–21, Listing 7.15) to which the calculateModelNames delegates (lines 8, 11, and 14).

By default, a calculateModelNamesFor* method returns a set which only contains the value of the parameter name. The language engineer can efficiently adapt the calculation of possible model names for a specific symbol kind by solely overriding the corresponding hook point method [Pre95a] (cf. Section 7.14).

7.11 Generation of Model Loaders

Each language provides its specific model loader in order to process models of that language. For this, the model loader only extends CommonModelLoader and implements the method createSymbolTableFromAST in a schematic way. Consequently, it mainly consists of boilerplate code (although language-specific information is required).

Boilerplate Code

A concrete model loader:

1. extends CommonModelLoader (cf. Section 6.9.3),
2. implements the method createSymbolTableFromAST inherited from CommonModelLoader and sets the root AST node's type as createSymbolTableFromAST's first parameter, and
3. calls the (language-specific) symbol table creator within createSymbolTableFromAST.

Generative Approach

While the first item concerns generic aspects the other two rely on language-specific information, that is:

- the name of the start production to determine the name of the top-level AST node (e.g., AST**Automaton**) (see 2.),

- the grammar name for determining the name of the symbol table creator class (e.g., **AutomatonDSL**`SymbolTableCreator`, cf. Section 7.9) (see 3.), and
- the grammar name for setting the model loader's class name (e.g., **Automaton-DSL**`ModelLoader`).

No further information is required for implementing a language-specific model loader (cf. Section 6.9.3), and thus, it can be completely generated.

7.12 Generation of Resolving Filters

In general, the default implementation for resolving filters, i.e., `CommonResolving-Filter` is sufficient, and hence, implementing language-specific resolving filter classes is not necessary (cf. Section 4.2.2). However, dedicated classes simplify reuse, for example, when composing languages. Thus, the generator produces a resolving filter class for each symbol kind.

Boilerplate Code

Resolving filter classes are very small and almost fully consist of boilerplate code (cf. Listing 7.16), that is:

1. extending `CommonResolvingFilter` and
2. implementing a constructor which passes the respective symbol kind to its superclass' constructor.

Generative Approach

The generator only needs the symbol's name (i.e., the name of the annotated production):

- as the resolving filter's name, e.g., **State**`ResolvingFilter` (line 1, Listing 7.16),
- in order to define its constructor (line 4) (see 2.), and
- to pass the correct symbol kind to its superclass' constructor, for example, **State-**`Symbol.KIND` (line 5) (see 2.).

```java
public class StateResolvingFilter
      extends CommonResolvingFilter {

  public StateResolvingFilter() {
    super(StateSymbol.KIND);
  }
}
```

Java «GEN»

Listing 7.16: Generated resolving filter class for states.

7.13 Generation of Modeling Language Configurations

The `ModelingLanguage` interface serves as a configuration that bundles a language's components, such as its parser, symbol table creator, and model loader (cf. Section 6.9.2). Since we generate each of these components at least partially (cf. Table 7.8 on page 176), we can also generate the language-specific modeling language class. Figure 7.17 exemplifies this with the architecture of `AutomatonDSLLanguage`. As it can be seen, all associated classes of `AutomatonDSLLanguage` are (at least) partially generated.

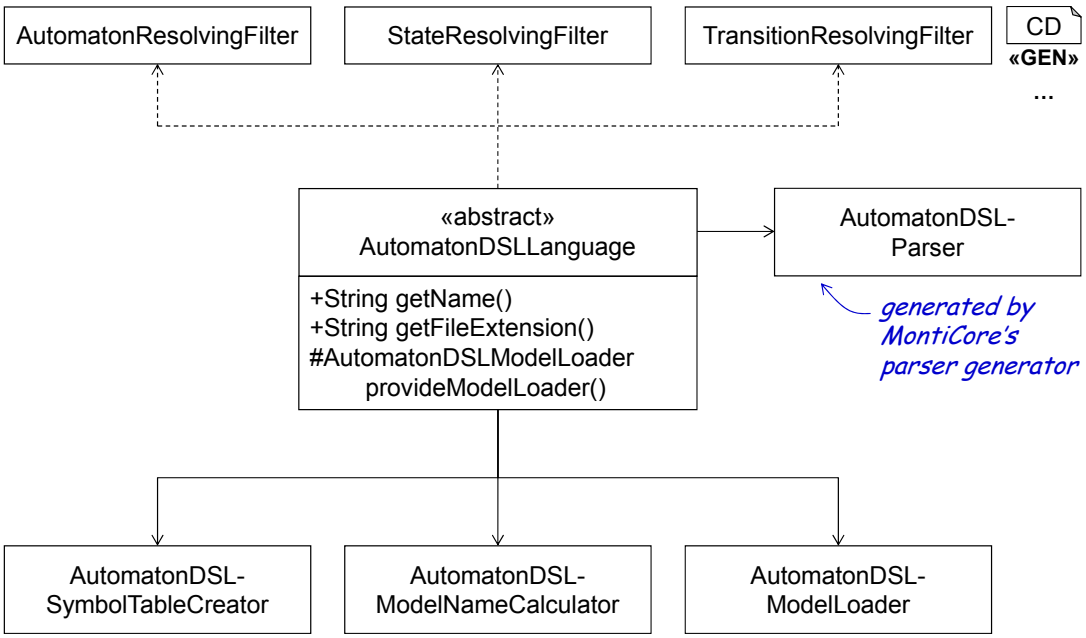


Figure 7.17: Overview of generated modeling language class and its associated classes.

Boilerplate Code

A language-specific modeling language:

1. subtypes `CommonModelingLanguage` (cf. Section 6.9.2),
2. overrides the respective methods to set the associations shown in Figure 7.17, and
3. overrides `provideModelLoader` for instantiating the corresponding model loader.

Generative Approach

Again, the first item is not based on language-specific information but integrates the modeling language into SMI. For the remaining points, the generator requires the following information:

- the grammar name for setting the modeling language's class name (e.g., **`AutomatonDSL`**`Language`),
- the grammar name for determining the name of the parser[4], the symbol table creator (cf. Section 7.9), the model name calculator (cf. Section 7.10), the model loader (cf. Section 7.11), and to set the return type of the corresponding methods (e.g., **`AutomatonDSL`**`ModelLoader provideModelLoader()`), and
- the name of the productions annotated with `@!` to initialize the resolving filters, e.g., `addResolvingFilter(new` **`State`**`ResolvingFilter())` (cf. Section 7.12).

The information left is the language's name (e.g., "Simplified Automaton Language") and its file extension which cannot be obtained from the grammar. As a result, `AutomatonDSLLanguage` is generated as an abstract class. Additionally, the generator produces the method `initResolvingFilters` to initialize language-specific resolving filters (line 7 and lines 11–15, Listing 7.18). This enables to efficiently customize the registered resolving filters via method overriding.

Finally, it is important to mention that the generator does not produce the method `provideModelLoader` (lines 17–20) in case a handwritten modeling language class exists. Otherwise, the keyword `this` (line 19) would yield a compilation error since it should refer to the handwritten class, not to the generated class.

7.14 Adapting the Generated Classes

Enriching the grammar with few symbol table information allows the generator to produce large parts of the language-specific symbol table infrastructure (cf. *GRQ1*). While some

[4] generated by MontiCore's parser generator (cf. Section 2.2.3)

Java «GEN»

```
1 public abstract class AutomatonDSLLanguage
2               extends CommonModelingLanguage {
3
4   public AutomatonDSLLanguage(String langName, String fileExt) {
5     super(langName, fileExt);
6
7     initResolvingFilters();
8     setModelNameCalculator(new AutomatonDSLModelNameCalculator());
9   }
10
11  protected void initResolvingFilters() {
12    addResolvingFilter(new AutomatonResolvingFilter());
13    addResolvingFilter(new StateResolvingFilter());
14    addResolvingFilter(new TransitionResolvingFilter());
15  }
16
17  @Override
18  protected AutomatonDSLModelLoader provideModelLoader(){
19    return new AutomatonDSLModelLoader(this);
20  }
21
22  ...
23 }
```

Listing 7.18: Excerpt from the generated `AutomatonDSLLanguage` class.

classes are completely generated, and thus, ready for use (such as the symbol kinds and resolving filters), others are partially generated and must be extended (e.g., symbols) (cf. *GRQ3*). Since the generated defaults do not always fit the requirements, the language engineer must have the opportunity to efficiently adjust them (cf. *GRQ2*).

The remainder of this section gives a brief description about how the language engineer can conduct customization of generated code using MontiCore's integration mechanisms (cf. [GHK+15b, HMSNR15b]).

MontiCore comes with an integration mechanism (based on the *Extended Generation Gap* [GHK+15b]) for handwritten and generated code in order to simplify the process of customizing generated classes and integrating them with handcoded ones. Furthermore, the mechanism allows to ignore generated classes without worrying about name clashes (cf. *GRQ4*). Figure 7.19 demonstrates how it works.

In the example depicted in Figure 7.19 the generator produces the two classes `TransitionSymbol` and `StateSymbol` whereby the former uses the latter, as shown in the left part. If the language engineer adds a handwritten class `StateSymbol` in the *same package* as the generated one, the names will clash and a compilation error will

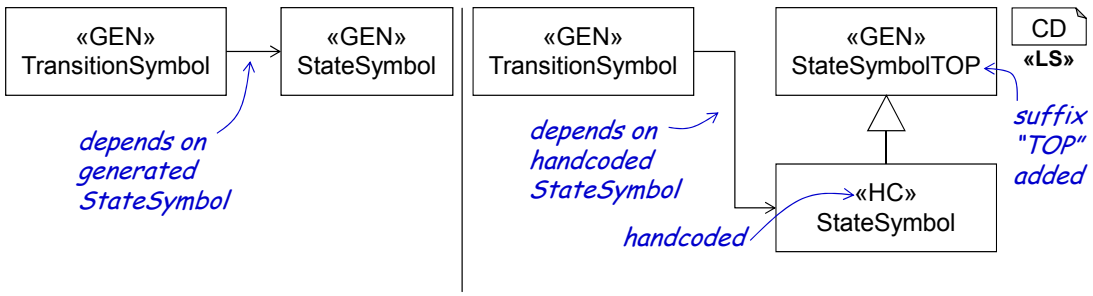


Figure 7.19: Generator is aware of handwritten classes to facilitate integration with generated code (cf. *Extended Generation Gap* [GHK+15b]).

occur. Therefore, when (re-)running the generator, it is aware of the handwritten class `StateSymbol` and produces a class `StateSymbol`**`TOP`** instead, which can be but need not be inherited.

All classes that depended on the previously generated `StateSymbol` automatically depend on the handwritten `StateSymbol` without any modification needed since it is defined in the same package as the generated class. On the one hand, this prevents name clashing between handwritten and generated code (important for realizing *GRQ3* and *GRQ4*). On the other hand, this mechanism enables efficient integration of handwritten and generated code (cf. *GRQ2*), following the *convention-over-configuration* approach.

Overall, the language engineer has the following four options for dealing with the generated code discussed throughout this chapter [HMSNR15b]:

1. Ideally, the generated language-specific symbol table infrastructure completely fulfills the requirements, and hence, is ready for use (or at least with very little effort). This (if at all) applies for languages that are not very complex, such as the automaton example used throughout this chapter. However, since a symbol table strongly depends on the language's semantics as well as the language engineer's design decision, it typically cannot be fully generated only with the information stated in the (annotated) grammar (cf. completeness in [VBD+13]).

2. The language engineer can completely ignore the generated classes and instead implement the language-specific symbol table from scratch (cf. *GRQ4*). However, in such a case it is not required to annotate the grammar with symbol table information.

3. The language engineer can use (parts of) the generated classes as one-shot generation. That is, she copies generated classes to the handcoded source path and modifies them as needed. From then on, she can ignore the generated classes as the integration mechanism prevents name clashing. As a consequence, changes in the grammar have no affect on the (previously generated) code anymore.

4. The generated infrastructure (or parts of it) can be customized and extended in a non-invasive way, using MontiCore's integration mechanism presented above as well as integration mechanisms for object-oriented languages discussed in [GHK+15b], such as the generation gap [Vli98, SV06, Fow10]. Unlike one-shot generation mentioned in the previous item, those integration mechanisms enable that grammar and generated code remain consistent.

These four options can be used intertwined for different parts or components of the symbol table infrastructure. Even if unused, they can serve as documentation and starting point for the manual implementation. Analogously to tests, they help to understand the symbol table infrastructure by demonstrating how language-specific classes must be integrated into the generic infrastructure SMI. Moreover, they illustrate how to adhere to the guidelines, patterns, and methods elaborated in Chapters 4, 5, and 6.

Chapter 8

Infrastructure for Language Composition

Several modeling languages can be integrated to specify complex software systems. Therefore, a mixture of DSLs and GPLs can be employed [CvdBCR15] to describe the different aspects of the system in an appropriate way. This is referred to as *language composition* where "multiple languages [work] together to achieve a common goal" [CvdBCR15].

The MontiWIS framework [RR13, Rei16], for example, provides a language family for developing web-based information systems in a model-based and generative way. Here, a *page description* language allows for specifying static web pages. *Activity diagrams* describe the business logic of the system and handle the linking of web pages. For more sophisticated behavior, *Java* can be embedded in activity diagrams.

Another example is the MontiArcAutomaton ADL [RRW13b] which extends the MontiArc ADL [HRR12] and additionally embeds a state-based component behavior modeling language. To specify data types, the MontiArcAutomaton ADL constitutes a language family with UML/P class diagrams [Sch12, Rum16].

This chapter focuses on syntactic language composition, and in particular, the composition of symbol tables (as part of the abstract syntax, cf. Section 3.8). It is based on MontiCore's syntactic composition mechanism (including concrete syntax, abstract syntax tree, and symbol table) elaborated in [Völ11] (cf. Section 2.2). For semantic language integration please refer to [GR11].

MontiCore allows for three types of language composition, namely language embedding, language aggregation, and language inheritance [Völ11, HLMSN+15a]. *Language embedding* enables a language to embed elements of other languages. In *language aggregation* models of heterogeneous languages are composed and interpreted as a whole. *Language inheritance* allows to extend or refine a language in sublanguages.

To enable models (or model elements) of a language L_2 to refer to models (or model elements) of a language L_1, the composition can be conducted in two ways:

$L_2 \xrightarrow{knows} L_1$ *(Language L_2 knows about L_1)*: In this case, L_2 directly uses components (e.g., symbols or AST nodes) of L_1. For example, if the OCL language is developed with the knowledge of the class diagram language, it can directly refer to its elements, such as classes and methods.

$L_2 \xrightarrow{knows\ not} L_1$ *(Language L_2 does not know about L_1)*: This case, among others, occurs when composing languages that were developed independently from each other (e.g., SQL and Java). In those cases, a third (glue) component conducts the language composition and often translates[1] elements of L_1 to elements L_2 can deal this.

While in the first case L_2 is coupled to L_1, in the second case both languages are completely independent. Coupling languages has the advantage that it eases the implementation process; no additional glue artifacts need to be implemented which also increases development efficiency. However, coupling languages yields some major disadvantages. L_2 cannot be implemented independently from L_1, e.g., by another language engineer. This complicates (distributed) working in teams. Also, changes in L_1 can affect L_2. Furthermore, L_1 cannot be exchanged easily, for example, to allow OCL to work with both class diagrams and object diagrams.

SMI allows for both cases, i.e., $L_2 \xrightarrow{knows} L_1$ and $L_2 \xrightarrow{knows\ not} L_1$. The remainder of this chapter mainly focuses on the latter case since it is more complicated and requires an integration of heterogeneous and independent languages [LNPR+13, HLMSN+15b].

8.1 Overview and Primary Requirements

Figure 8.1 highlights language embedding and language aggregation based on the (generic) scope graph. In the example, a language L_2 (or parts of it) is embedded into a language L_1. Since language embedding results in one monolithic model (or artifact), it only concerns a subgraph of a model's symbol table. The resolution mechanism must be aware of this language switch during the bottom-up (as well as top-down) intra-model process (cf. Section 6.2). Language aggregation composes the models of heterogeneous languages by their names and kinds (same as model composition within a single language, cf. Chapter 6). Hence, models of L_1 and L_3 in Figure 8.1 remain separated. For this, the inter-model resolution process must enable access to models of either language and also handle the language switch. Concerning the scope graph, language inheritance results in a new (single) language. Consequently, its respective scope graph is same as for a single language (cf. Chapter 6) and does not require further translations.

Although three different languages are involved, the emerged scope graph depicted in Figure 8.1 does not differ from the scope graph of a single language, as presented in, for example, Chapter 6. Only additional translations are required. This emphasizes the importance of the *generic* scope graph which allows for composition of models from heterogeneous languages (cf. Section 4.2.2).

From the general proceeding highlighted in Figure 8.1, we derive the leading requirements for the language composition infrastructure:

[1] Section 8.5 discusses a case where no translation is required for the composition.

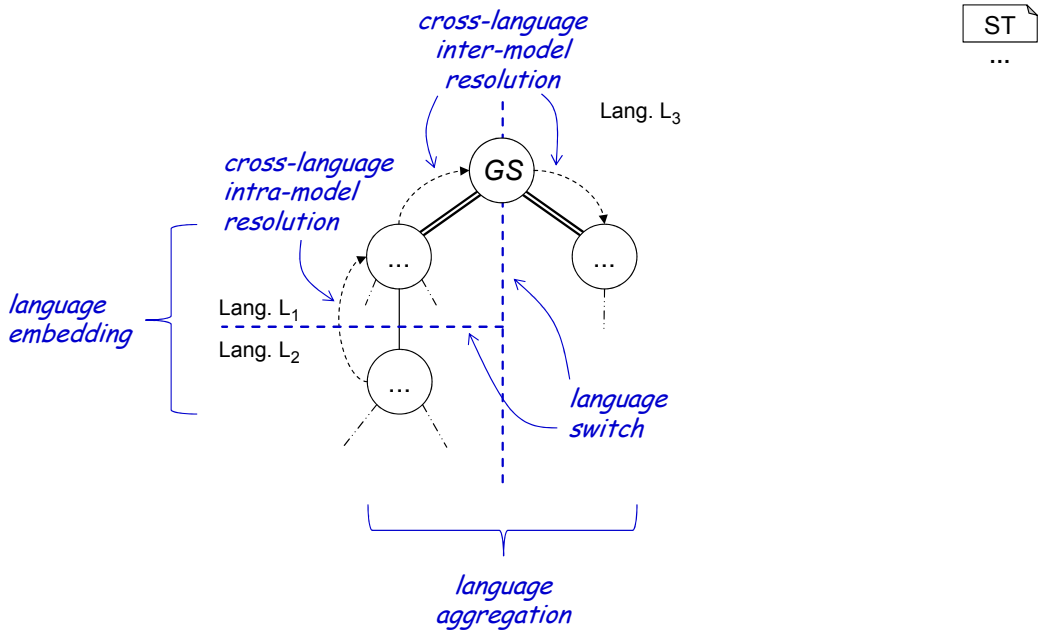


Figure 8.1: Overview of a scope graph resulting from language composition. The scope graph itself is same as for single languages. In contrast to language inheritance, language embedding and language aggregation require translations between elements of the involved languages.

CRQ1 (Reuse) The resolution mechanism introduced in Chapter 6 should be completely reused for language composition. In particular, there should not exist a second structure only for language composition. This eases the learning curve for both symbol table engineers and users since they do not need to learn further concepts[2].

CRQ2 (Open for Extension/Closed for Modification) With respect to composition, each language "should be open for extension, but closed for modification" [Mey88], following Meyer's open/closed principle. That means, the composition should only require *additional* configuration without modifying existing languages.

CRQ3 (Efficiency of Configuration) Given different languages their composition should be configured with *minor* adaptations.

CRQ4 (Composition Without Translation) Composing two languages L_1 and L_2 where L_2 explicitly uses elements of L_1 (case $L_2 \xrightarrow{knows} L_1$) does not require a translation from L_1's elements to L_2's elements (see above). This should also apply

[2]Chapters 4, 5, and 6 already discuss design decisions that enable language composition with SMI.

for sublanguages of L_1. More precisely, once composed with L_1, L_2 should be able to employ any element of L_1's sublanguages *without* translations (additional configuration might be required).

CRQ5 (Language-Specific Usage) Languages should be composed in an encapsulated way so that the user of a single language does not need to know about the composition. This is important for an a-posteriori composition of languages [HLMSN+15a].

Chapter Outline

Sections 8.2, 8.3, and 8.4 (i) introduce language embedding, language aggregation, and language inheritance, respectively, (ii) elaborate their technical realization via SMI, and (iii) discuss related approaches. Next, Section 8.5 presents a generic symbol table infrastructure for Java-like languages. Section 8.6 compares transitive and non-transitive translations between elements of different languages. Subsequently, Section 8.7 demonstrates a more complex example of language composition. Section 8.8 finally discusses some alternative classifications of language composition.

8.2 Language Embedding

In language embedding a host language L_H embeds elements of (at least) one language L_E, resulting in a new language L_{HE}. Although models of L_{HE} combine elements of both languages L_H and L_E in the same artifact, these languages are still developed independently. This requires the languages to be composed in a non-invasive manner (cf. *CRQ2*).

Language embedding employs grammar embedding which, as described in Section 2.2.2, results in a composed AST structure where a (sub-)tree of L_E's AST is attached to a node of L_H's AST [HLMSN+15a, HLMSN+15b]. If the embedded AST nodes represent essential aspects (cf. Section 3.8), language embedding also includes composition of the respective ST elements. Figure 8.2 conceptually illustrates the idea.

The left part of Figure 8.2 depicts the AST and the corresponding ST of L_H (top) and L_E (bottom). Only the root node and the node N_k of L_E are related to elements of symbol table ST_E, in contrast to, e.g., N_i. Embedding N_i and N_k in AST_H results in the composed AST structure depicted on the right part of Figure 8.2. While N_i does not affect the respective scope graph, embedding N_k leads to the embedding of its related symbol table element into ST_H. This thesis particularly focuses on cases like N_k where AST embedding also yields ST embedding. Please refer to [Völ11] for an elaboration on AST embedding in MontiCore. A brief summary can be found in Section 2.2.2. To enable language embedding for the respective symbol tables, we employ SMI's *generic* scope structure, as shown in the remainder of this section.

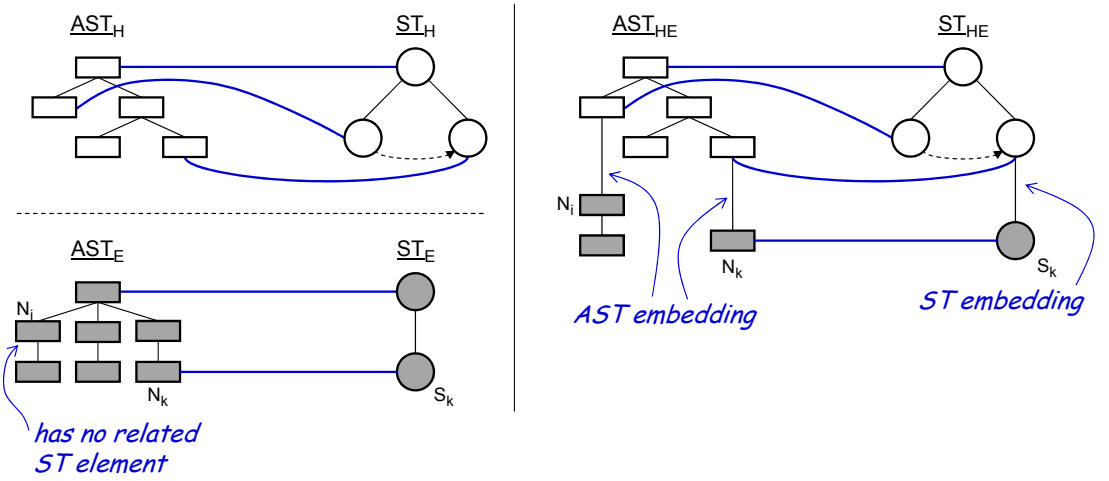


Figure 8.2: Conceptual idea of language embedding including embedding of ST elements.

8.2.1 Example

This section demonstrates language embedding by the example of the two *independent* languages SQL and Java (cf. [Ora13]). SQL [SQL11] is a language for database management. Figure 8.3 shows an example where a SQL statement is embedded in a Java method (line 3) to request the name of the user with the id `uid`. The construct `sql:[...]` ensures parseability (since Java does not allow such constructs).

For this example we assume that the corresponding SQL table is as shown in Figure 8.3 (bottom part), i.e., the SQL table *Users* exists and consists of the two columns *Name* and *UserId*. For the mapping between SQL variables and Java parameters a naming convention is applied. That is, a SQL variable name (e.g., `"uid"` in line 3) corresponds to the Java parameter name (e.g., `"id"` in line 1) with the additional prefix `"u"`.

The glue grammar `JavaWithSQL` shown in Listing 8.4 conducts grammar-based embedding of Java and SQL. First, `JavaWithSQL` extends both grammars (line 1). Next, it defines the nonterminal `JavaSQLExpression` which implements `JavaExpression` (line 4), i.e., an interface nonterminal of the Java grammar, and assigns the nonterminal `SQLQuery` of the SQL grammar to it (line 5).

Given the above mentioned naming conventions between Java parameters and SQL variables, we can *statically* check the well-formedness of the embedded SQL statement based on the parameter `id` (assuming the SQL table is up-to-date). The SQL variable `uid` really refers to the `id` parameter defined in the enclosing Java method. Please note that although knowledge of the *Java parameter* (i.e., `JavaParamSymbol`) is required to specify the correct SQL variable, the respective SQL language itself expects a *SQL variable* (i.e., `SQLVariableSymbol`). Consequently, the languages are still separated technically (cf. Section 8.2.2).

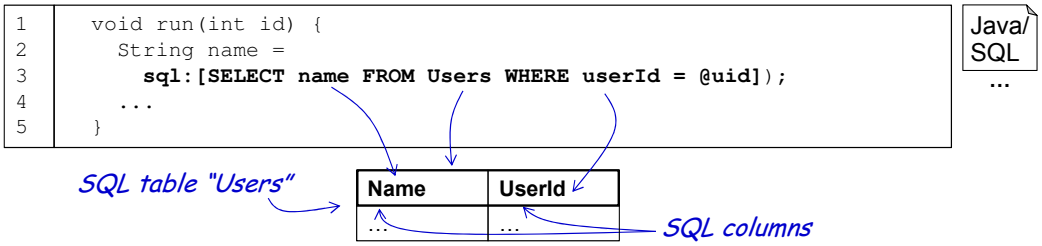


Figure 8.3: Exemplary model for Java with embedded SQL.

```
1 grammar JavaWithSQL extends Java, SQL  {                    MCG
2
3   // Enables SQL queries in Java expressions
4   JavaSQLExpression implements JavaExpression =
5       "sql:[" SQLQuery "]";
6 }
```

Listing 8.4: Glue grammar for Java with embedded SQL.

Figure 8.5 presents the emerging scope graph of the composed model shown in Figure 8.3. As it can be seen, the scope graph enlightens *the essence of the combined model*. That is, the embedded SQL results in a subgraph of the enclosing `run` method scope. Moreover, a name used in SQL refers to the respective Java element. Since Java and SQL are two independent languages (case $L_2 \xrightarrow{knows\ not} L_1$), the following problems occur:

- When starting the resolution for SQL variables, e.g., via `resolve("uid", SQL-VariableSymbol.KIND)`, Java parameters are not considered since they do not constitute a kind hierarchy with the SQL variables (cf. Section 3.3).
- Even if Java parameters would be considered, e.g., via `resolve("uid", Java-ParamSymbol.KIND)`, the class `JavaParamSymbol` is unknown in the SQL language, and thus, cannot be handled in that language.

In case the SQL language knew about Java parameters (case $L_2 \xrightarrow{knows} L_1$), the complete resolution process elaborated in Chapter 6 could be reused *without further adaptions* (cf. *CRQ1*). For this, SQL would have to explicitly use elements of the Java language which, however, comes with the price of higher coupling between these languages.

The next section elucidates how the above mentioned problems are eliminated with a translation process during the resolution, especially for case $L_2 \xrightarrow{knows\ not} L_1$.

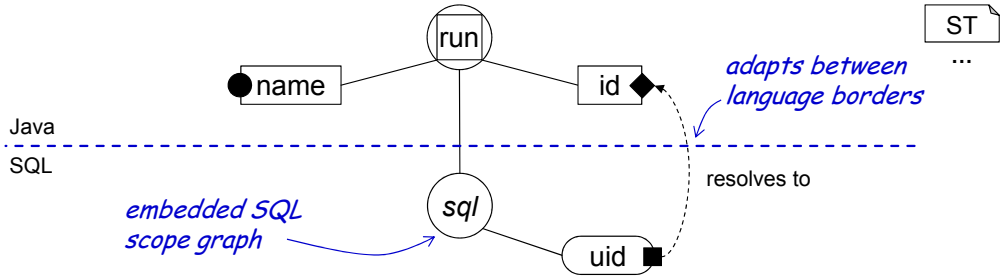


Figure 8.5: Emerging scope graph of model in Figure 8.3.

8.2.2 Cross-Language Intra-Model Resolution

To enable usages of Java symbols within SQL, a translation from Java parameters to SQL variables (i.e., `JavaParamSymbol` to `SQLVariableSymbol`) is required. Moreover, a SQL resolution request must be translated to a respective Java resolution request. For example, resolving a SQL variable must ultimately resolve a Java parameter. In summary, the following two translations are required (cf. [Völ11]):

Translation of Resolution Request: Symbols are resolved via their name and kind (cf. Chapter 6). Hence, a translation from a kind k_2 of language L_2 to a kind k_1 of language L_1 is needed, if resolving a symbol of kind k_2 should result in a symbol of kind k_1. For the above example this means that requests like `resolve("uid", SQLVariableSymbol.KIND)` have to be translated to `resolve("uid", JavaParamSymbol.KIND)`. To enable mappings via name convention, the names must be translated as well. For example, `resolve("uid", SQLVariableSymbol.KIND)` must be translated to `resolve("id", JavaParamSymbol.KIND)`.

Translation of Resolved Symbol: Next, if a symbol of language L_1 is found, it has to be translated to a symbol that language L_2 can handle. For example, the SQL language cannot access `JavaParamSymbol` directly. Instead, `JavaParamSymbol` has to be translated to `SQLVariableSymbol`, which includes a name translation (i.e., `"u" + name`), e.g., from `"id"` to `"uid"` (see below).

Resolving filters—as introduced in Section 4.2.2—play an essential role in realizing the aforementioned translations. The main task of a resolving filter described so far is, given a set of symbols, it returns the symbols that yield a matching kind. Consequently, a symbol can only be resolved if a corresponding resolving filter for its kind is registered. To enable language embedding (and also language aggregation, cf. Section 8.3), this concept is extended with *adapted resolving filters* which conduct the two translations. For this, SMI provides the interface `AdaptedResolvingFilter` and its default implementation `CommonAdaptedResolvingFilter`, as illustrated in Figure 8.6.

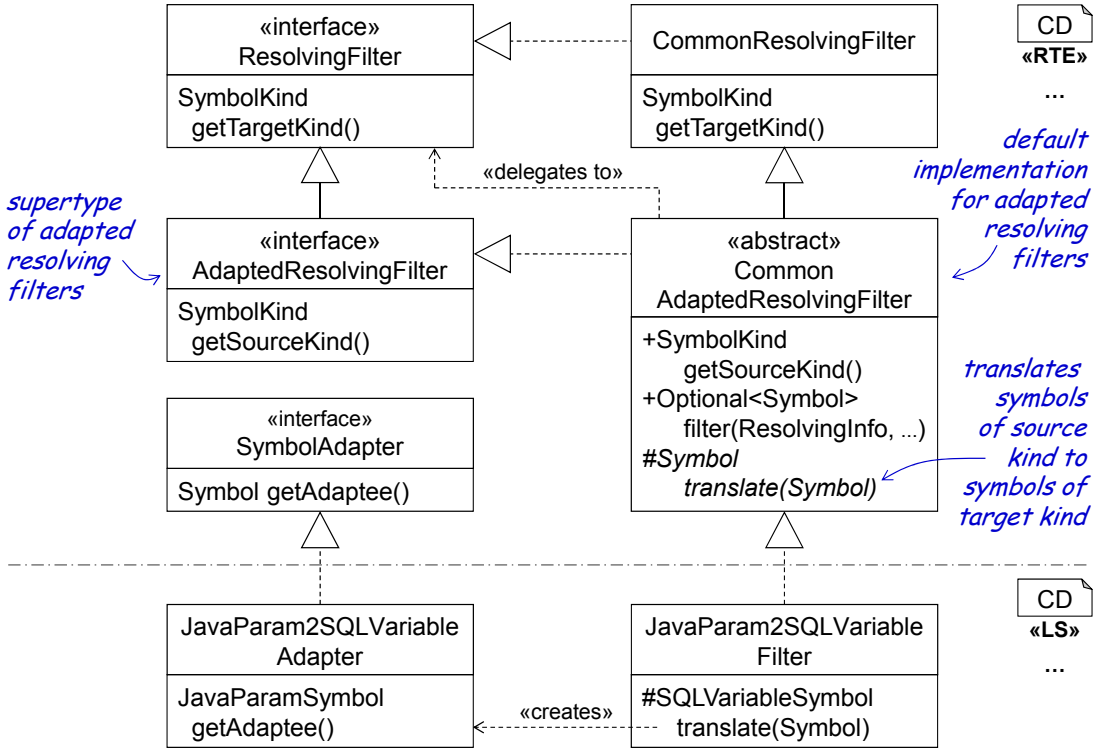


Figure 8.6: Infrastructure for adapted resolving filters.

Same as regular resolving filters, adapted resolving filters have a target kind which is the kind of symbols they return. Therefore, `AdaptedResolvingFilter` extends `ResolvingFilter`. Additionally, the method `getSourceKind` specifies the kind of symbols that are translated to the target kind. Moreover, `CommonAdaptedResolvingFilter` provides the factory method [GHJV95] `translate` as hot spot [Pre95a, Pre95b] for specifying the translation of the resolved symbol applying the Adapter pattern [GHJV95].

As exemplified in Figure 8.7, `JavaParam2SQLVariableAdapter` translates a `JavaParamSymbol` to a `SQLVariableSymbol`. For this, `JavaParam2SQLVariableAdapter`, among others, overrides `getName` to translate a Java parameter name to a SQL variable name according to the above mentioned naming convention.

Finally, the `filter` method of `CommonAdaptedResolvingFilter` (cf. Figure 8.6) conducts a translated filtering by delegating to resolving filters (retrieved via the `ResolvingInfo` parameter) whose target kind match the filter's source kind. `filter` also conducts the name translation if necessary, e.g., changing `"uid"` to `"id"`.

Figure 8.8 demonstrates the translation process by the example of the resolution of the SQL variable `uid` (cf. Figure 8.5). The resolution starts within the SQL scope via `resolve("uid", SQLVariableSymbol.KIND)` and proceeds as follows (cf. [Völ11]):

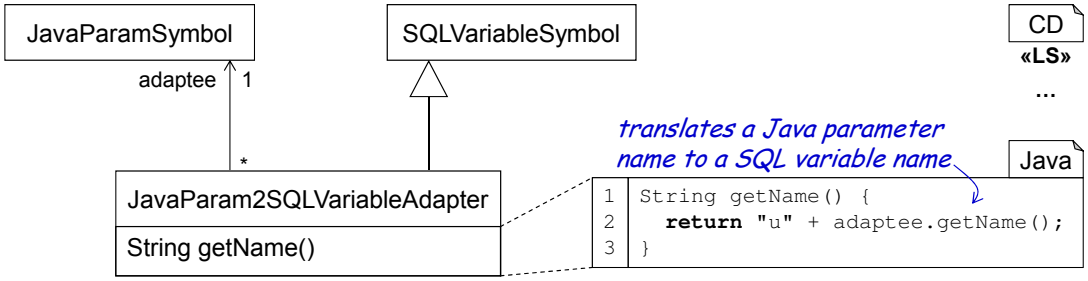


Figure 8.7: Adapter for translating a Java parameter symbol to a SQL variable symbol.

- It starts a local search in the SQL scope and continues in `run`'s spanning scope.
- There, it again conducts a local search, considering all resolving filters with *target kind* `SQLVariableKind`. This applies for, among others, `JavaParam2SQLVariableFilter`, hence, `filter("uid", symbols)` is invoked on it.
- Next, `JavaParam2SQLVariableFilter` delegates to all resolving filters matching its source kind `JavaParamKind`, and changes the requested name to `"id"`.
- The regular filter `JavaParamFilter` then finds the respective parameter symbol `id` and returns it to the `JavaParam2SQLVariableFilter` object.
- `JavaParam2SQLVariableFilter` finally translates `id` to a `SQLVariableSymbol` via `translate(id)` and `JavaParam2SQLVariableAdapter`.

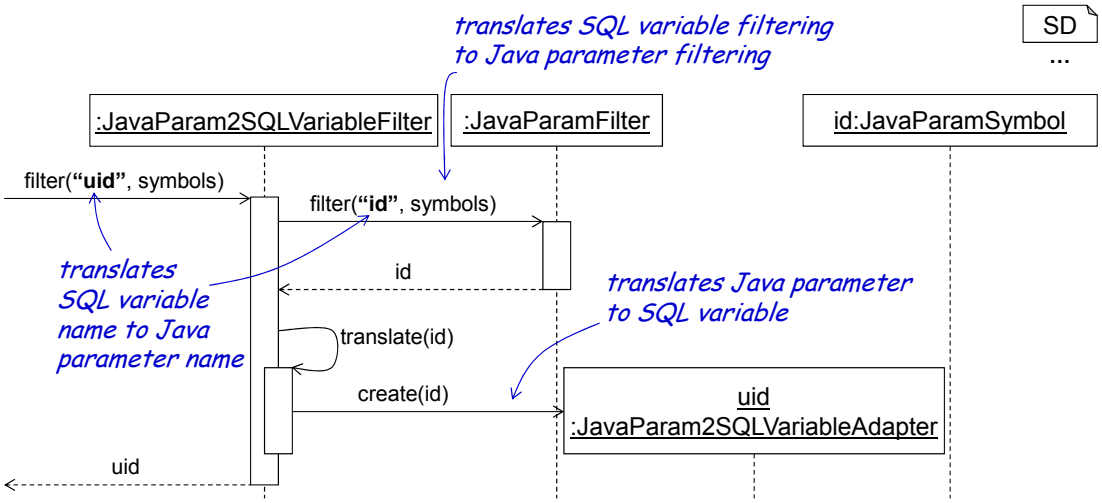


Figure 8.8: Exemplary process for translating a Java parameter symbol to a SQL variable symbol via resolving filters.

As shown in the example, the task of `JavaParam2SQLVariableFilter` is to find the SQL variable symbol `uid` by finding the corresponding Java parameter symbol `id` (via delegation) and finally adapting it to a SQL variable symbol.

8.2.3 Symbol Table Creator for the Composed Language

Since language embedding allows for composed models that contain (syntactic) elements of the host language as well as its embedded languages, both the parsing process and the symbol table creation process must be aware of these different elements when (re-)loading models from artifacts. Briefly speaking, the parsers and the symbol table creators of the languages must be composed. In MontiCore, the parser generator is aware of this and generates a composed parser based on the glue grammar (cf. Section 2.2.2). A symbol table creator is a visitor that traverses the AST in order to build up the scope graph (cf. Chapter 5). Technically, this means that a language-specific symbol table creator must subtype `SymbolTableCreator` as well as one of the generated language-specific visitors (cf. Section 2.2.4). Ideally, a language engineer employs the delegator visitor to reuse each language's symbol table creator when applying language embedding.

Figure 8.9 demonstrates the symbol table creator architecture of the Java language with embedded SQL statements. The `JavaWithSQLSymbolTableCreator` class represents the symbol table creator and extends `CommonSymbolTableCreator`. Furthermore, it implements the generated default visitor `JavaWithSQLVisitor` which itself extends the default visitors of the two languages, namely `JavaVisitor` and `SQLVisitor`. That way, `JavaWithSQLSymbolTableCreator` enables traversing the AST of the composite language. In order to apply the correct operations on the specific AST nodes, `JavaWithSQLSymbolTableCreator` makes use of the delegator visitor which delegates to `JavaSymbolTableCreator` or `SQLSymbolTableCreator`, depending on the AST node type (cf. Section 2.2.4).

The initialization of the `JavaWithSQLDelegatorVisitor` takes place in the constructor of `JavaWithSQLSymbolTableCreator`, as shown in Listing 8.10. First, the symbol table creator of the Java language is created (lines 6–7). By passing the `scopeStack` (inherited from `CommonSymbolTableCreator`) to the constructor, it is ensured that all symbol table creators use the same stack. This is important, to enable that symbols of the SQL language can be added to scopes of the Java language. The algorithm for building up the symbol table as presented in Section 5.3 remains unchanged. Also, the functionality provided by `CommonSymbolTableCreator` as introduced in Section 5.5 is fully reused. The initialization of `SQLSymbolTableCreator` is conducted analogously to `JavaSymbolTableCreator` (lines 8–9). It is important that both symbol table creators provide the respective constructor of their superclass, i.e., `CommonSymbolTableCreator(ResolvingConfiguration, Deque<MutableScope>)` (cf. Section 5.5). That way, the symbol table creators share the same scope stack, enabling the stack-based symbol table creation approach presented in Section 5.3.1.

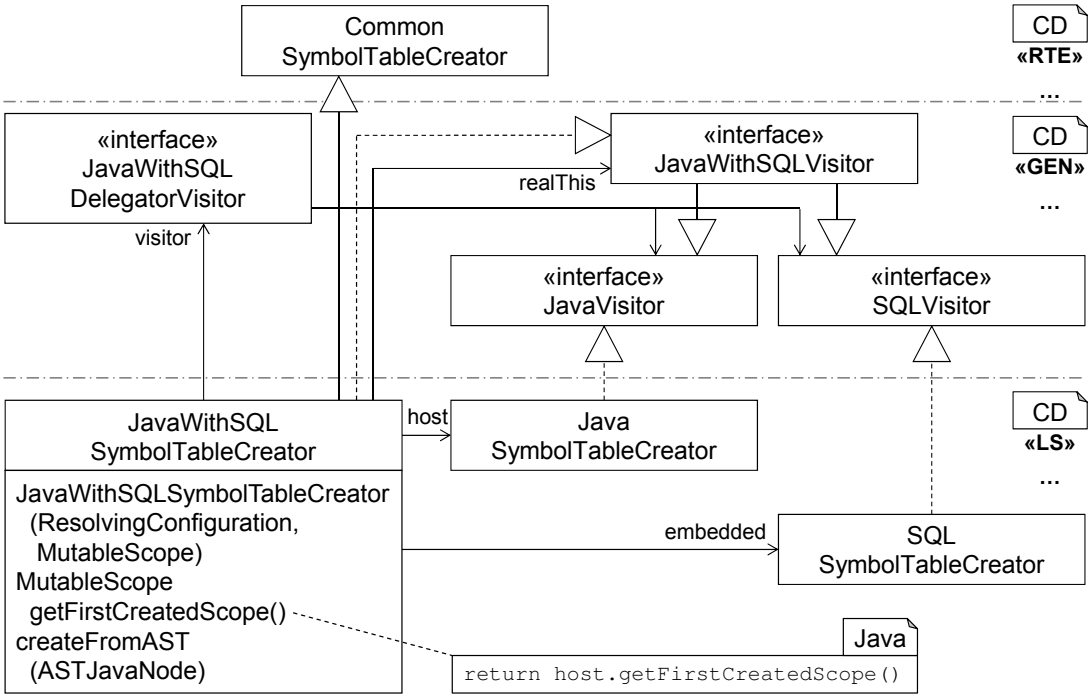


Figure 8.9: Overview of the symbol table creator structure for Java with embedded SQL.

```
1 public JavaWithSQLSymbolTableCreator(
2           ResolvingConfiguration resConfig,
3           MutableScope enclosingScope) {
4   super(resConfig, enclosingScope);
5
6   this.javaSTCreator =
7       new JavaSymbolTableCreator(resConfig, scopeStack);
8   this.sqlSTCreator =
9       new SQLSymbolTableCreator(resConfig, scopeStack);
10
11  visitor = new CommonJavaWithSQLDelegatorVisitor();
12  visitor.set_JavaWithSQLVisitor(this);
13  visitor.set_JavaVisitor(this.javaSTCreator);
14  visitor.set_SQLVisitor(this.sqlSTCreator);
15 }
```

Java «HC»

Listing 8.10: Implementation of a symbol table creator for Java with embedded SQL.

Finally, an instance of the generated `CommonJavaWithSQLDelegatorVisitor` (line 11) is initialized with the symbol table creator of the composite language, i.e., the `JavaWithSQLSymbolTableCreator` (line 12), and the symbol table creator of each single language, i.e., `JavaSymbolTableCreator` (line 13) and `SQLSymbolTableCreator` (line 14). For this, each of these symbol table creators has to apply MontiCore's *realThis* pattern (cf. Section 2.2.4), i.e., `this` must be omitted for shared data (e.g., the scope stack). Instead, `setRealThis` and `getRealThis` are to be used.

Same as for `JavaSymbolTableCreator` and `SQLSymbolTableCreator`, `JavaWithSQLSymbolTableCreator` must provide the above mentioned constructor (not shown in Listing 8.10) to enable composition with other symbol table creators.

8.2.4 Language Embedding Configuration

Composing languages via embedding emerges to a new (composite) language. Therefore, the configuration is similar to the configuration of a single language (cf. Section 6.9.2), i.e., besides several resolving filters, the new language has, among others, a model name calculator, a model loader, a (composed) symbol table creator, and a file extension.

SMI provides the class `EmbeddingModelingLanguage` to configure language embedding. As depicted in Figure 8.11, `EmbeddingModelingLanguage` subclasses `CommonModelingLanguage` and further provides a `host` language and several `embedded` languages to enable embedding of one or more languages. The bottom part of Figure 8.11 shows the configuration of the Java and SQL example described above. Each language has its own configuration, i.e., `JavaLanguage` and `SQLLanguage`, respectively. `JavaWithSQLLanguage` reuses them to configure their composition. By default, the model name calculator is the same as for the host language since (embedded) inner elements do not affect the enclosing model's name (cf. Section 6.9.5). The composed parser (generated from the grammar, not shown in Figure 8.11) as well as the composed symbol table creator (i.e., `JavaWithSQLSymbolTableCreator`, cf. Section 8.2.3) must be explicitly set in the language configuration. Also, the file ending changes (in most cases), e.g., *javasql* can be used instead of *java* to distinguish between the respective artifacts.

Furthermore, `EmbeddingModelingLanguage` unifies the resolving filters of the host language and its embedded languages. Therefore, `JavaWithSQLLanguage` only needs to add the adapted resolving filter `JavaParam2SQLVariableFilter`.

8.2.5 Discussion and Related Work

As already described in the introduction of Section 8.2, language embedding does not necessarily include symbol table embedding. It rather depends on whether the composed AST nodes (or nonterminals) embody essential model elements that are represented in the symbol table. If not, the composed language (i.e., L_{HE}) can neither define symbols of the embedded language(s) nor use them.

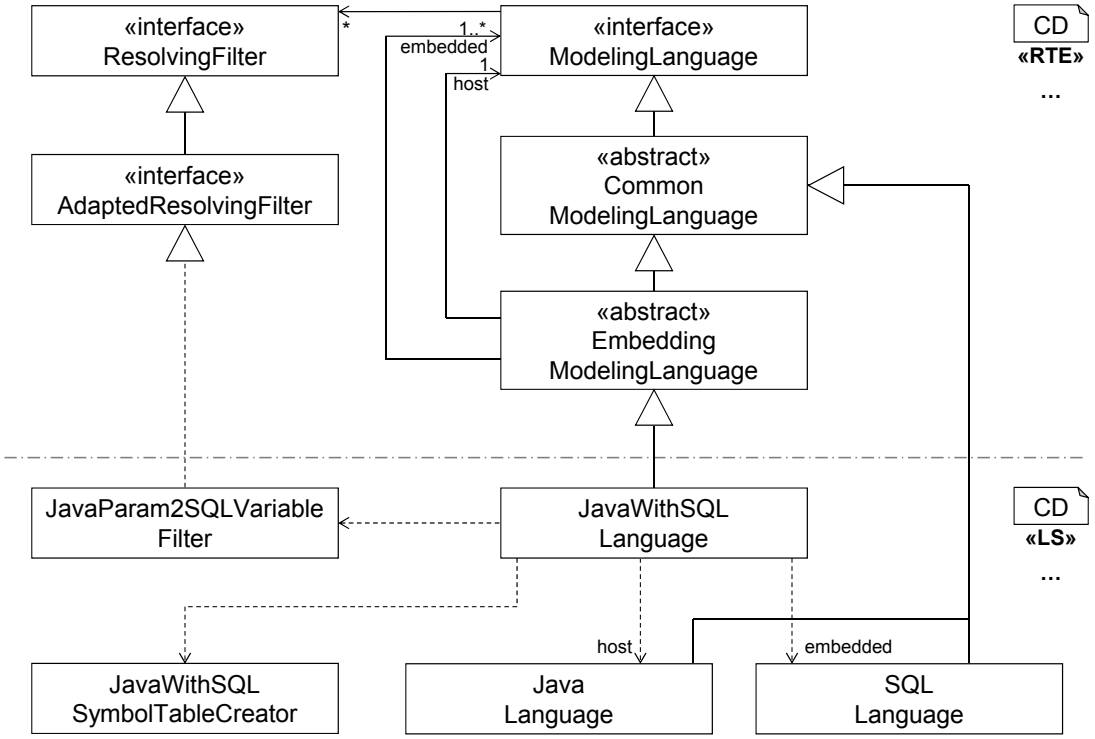


Figure 8.11: Infrastructure for language embedding.

In case the symbol tables are composed as well, the advantage of separating the generic scopes from their language-specific spanning symbols becomes apparent (cf. Section 4.3). The generic scopes allow for storing any kind of symbols including symbols of other languages (via method `add(Symbol)` of `MutableScope`, cf. Section 4.2.1). For example, SQL variables can be defined in a Java method scope via `methodScope.add(sqlVar)`. Resolving all Java variables (including local variables and parameters) via `methodScope.resolveLocally(JavaVariableEntry.KIND)` will also return all SQL variables defined in the method scope if an adapted resolving filter from SQL variables to Java variables is registered.

Since a (language-specific) symbol delegates to its spanned scope to retrieve the contained symbols (cf. Section 4.3), the translation mechanism discussed in Section 8.2.2 is fully reused, including name qualifying and model loading (cf. *CRQ1*). In the above example, `methodSymbol.getVariables()` returns all variables including SQL variables (adapted to Java variables). Hence, from a language user's viewpoint no knowledge about the composition is required (cf. *CRQ5*). This is essential since it ensures that, e.g., context conditions employing SMI proceed correctly in the single language as well as in a composition without the need for modification (cf. *CRQ2*).

The generic aspect of our approach is similar to Völkel's [Völ11] where generic namespaces are composed. However, in [Völ11] namespaces are not spanned by the respective symbol table entries, i.e., the concept of scope spanning symbols as in our approach does not explicitly exist as part of the infrastructure (cf. Section 4.3). As a consequence, accessing information via a symbol table entry can differ from information retrieved via its (spanned) namespace; the former does not consider translations while the latter does. Hence, in order to ensure consistency especially when composing languages, the language user has to rely on the *generic* namespaces and omit the *language-specific* symbol table entries. For example, only a generic access like `resolver.resolve("n", "Var", nsp)` includes entry translation, and thus, enables language embedding. In contrast, the composition infrastructure of the current thesis allows for a *language-specific* access like `type.getVarByName("n")` including the whole symbol translation process. Similar to namespaces in [Völ11], the generic scopes of SMI enable composition with scopes of other languages. That way, symbol table creators of the composed languages can still employ the stack-based approach discussed in Section 5.3 using visitor composition (cf. Section 2.2.4).

A symbol reference as presented in Section 4.4 (and defined in Def. 3.3) finds its respective symbol definition by starting the resolution process in its enclosing scope. This enables the symbol reference to reuse the whole resolution and symbol translation process as well. Hence, no configuration or glue component is required for symbol references (cf. *CRQ3*). In contrast, in Völkel's approach [Völ11] an adapted `IQualifiedEntryHandler` (cf. Section 4.4) is required which substitutes references with respective adapters. Furthermore, an `IQualifierClient` must be implemented for each *adapted* symbol table entry in order to qualify it. This, certainly, requires the language engineer to understand this additional concepts in order to apply them. SMI liberates the language engineer from these concepts. Instead, the language engineer only must specify the required translations (e.g., via `JavaParam2SQLVariableFilter`), SMI then takes over all the rest. Similar to adapted resolving filters, Völkel's approach requires adapted resolvers for translating the symbol table entries.

The approach in this thesis ensures that every symbol in the scope behaves the same (cf. Sections 3.5.3 and 4.2), even when composing languages. For instance, a SQL variable defined in a Java method is resolved same as a Java variable defined in that method, i.e., starting in the innermost scope, continuing with the class scope, supertypes, and so on (cf. Chapter 6). The adapted resolving filters *do not change* the resolution process but only *translate* the symbols defined in the scopes. In contrast, a resolver in [Völ11] specifies (to a high degree) the resolution process of a specific symbol table entry. Therefore, embedding an entry into a namespace that has a shadowing ability (as shadowing scopes in this thesis) might not affect the embedded entries, leading to different behavior.

In Xtext [Bet13], language embedding can be conducted as part of language inheritance (cf. Section 8.4) which, however, requires explicit knowledge of the extended language (case

$L_2 \xrightarrow{knows} L_1$). This is because Xtext does not allow for multiple grammar inheritance, and hence, it is not possible to employ a glue grammar as in MontiCore (cf. Section 2.2.2). As a consequence, Xtext does not allow for embedding of two (or more) *independent* languages (case $L_2 \xrightarrow{knows\ not} L_1$) [VBD+13].

Spoofax [KV10] conducts language embedding similar to the approach presented in the current thesis. Analogously to a glue grammar (cf. Section 2.2.2), an additional syntax definition module (defined via SDF [Vis97]) is required which imports the syntax definition modules of the host language and the embedded languages. Hence, embedding works for *independent* languages (case $L_2 \xrightarrow{knows\ not} L_1$). The module then defines new rules where it embeds elements of the embedded language into the host language. To prevent name clashes of sorts (i.e., terminals and nonterminals), Spoofax allows for different renaming strategies. Language embedding in Spoofax does not require to explicitly connect the name bindings of the respective languages. Technically this means, that no additional NaBL [KKWV13] models need to be created. Instead, the embedded elements only occur as elements of the host languages. This can require a type mapping between these elements via Spoofax's type specification language.

MPS [VS10] realizes language embedding via language inheritance (cf. Section 8.4.4) in combination with containment relations between concepts and that way enables case $L_2 \xrightarrow{knows\ not} L_1$. More precisely, an additional language extends L_1 and adds concepts of L_2 as children of the new subconcepts.

Paul Hudak [Hud98] introduces *domain-specific embedded languages* (DSELs) which are expressed with constructs of an existing programming language (such as Haskell or Java). That means, DSELs reuse both syntax and semantics of the host language (e.g., cf. [HORM08]), leading to the term *pure embedding*. In contrast, in this thesis the embedded DSL exists independently from its host language and comprises its own syntax and semantics. Fowler refers to this kind of DSL as *external DSL* and calls DSELs *internal DSLs* [Fow10] . External DSLs enable domain-specific error messages which are not possible with internal DSLs since they are limited to the constructs of the host language. An advantage of internal DSLs over external DSLs is that they allow for reuse of the host language's tool infrastructure, and thus, can (initially) be developed in an efficient way.

Erdweg et al. [EGR12] omit the notion of language embedding (as used in this thesis) since it can be realized with other approaches, such as language extension (language inheritance in our terms, cf. Section 8.4). Instead, they use the term self-extension where embedding of a language into a host language is specified with the host language itself[3]. In other words, the host language provides ways to be embedded with other languages. For this, Erdweg et al. in particular present string embedding and pure embedding (i.e., DSELs or internal DSLs). In *string embedding* the host language embeds another

[3]Hence, self-extension is a property of a language itself, not of the provided tool, such as MontiCore.

language using a string representation. In order to process (e.g., parse) those strings, the embedded language provides an API which the host language employs. Same as pure embedding, string embedding does not enable static analyses (e.g., context condition checks) in a proper way. Furthermore, string embedding is error-prone since arbitrary strings can be used. In contrast to [EGR12], we explicitly separate language embedding from language inheritance since the latter always considers the case $L_2 \xrightarrow{knows} L_1$. The former, however, also allows for $L_2 \xrightarrow{knows\ not} L_1$, and hence, requires additional glue code. Erdweg et al. further introduce the term *language unification* where "either language should be able to interact with [...] the other language" [EGR12]. This corresponds to our embedding approach (in combination with inheritance) in cases the languages know about each other, i.e., $L_2 \xrightarrow{knows} L_1$ and $L_1 \xrightarrow{knows} L_2$.

8.3 Language Aggregation

Language aggregation incorporates name-based model composition [Rum13, HR13] extended beyond language boundaries. In particular, this means that models of a language L_1 and a language L_2 can mutually refer to each other. These languages are said to constitute a *language family* [Völ11, HLMSN+15a]. In contrast to language embedding, the models are still managed in different artifacts, facilitating loose coupling [HLMSN+15a]. Same as for single languages the composed models of heterogeneous languages are interpreted together to describe different aspects of the software system [HLMSN+15a].

As shown in Figure 8.12, language aggregation does not manipulate the models' ASTs since the composition is conducted based on names. Hence, the respective ASTs nodes

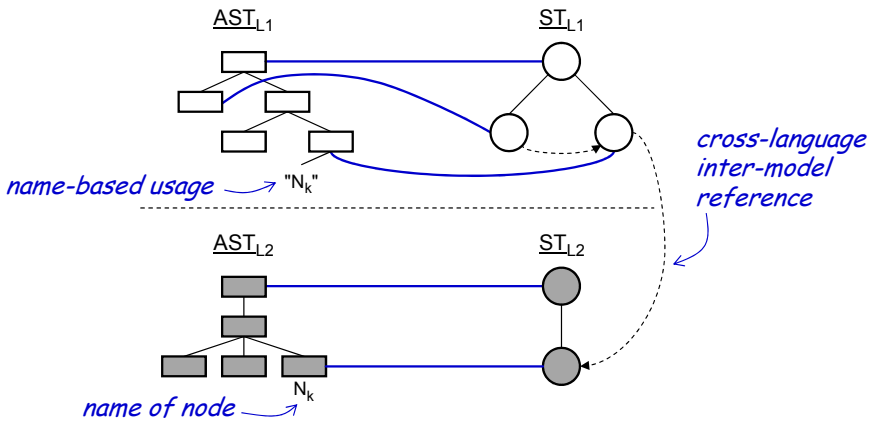


Figure 8.12: Conceptual idea of language aggregation which always includes aggregation of ST elements.

only hold *names* (as strings) of the referred model elements (left part, Figure 8.12). The names (and hence, the references) are ultimately resolved in the scope graph (right part, Figure 8.12). Consequently, language aggregation always includes aggregation of the respective symbol tables. In contrast, no grammar composition is required since elements of different languages are not intertwined.

8.3.1 Example

A prominent example of language aggregation is the UML language family [OMG15c]. It, among others, defines the object-constraint language (OCL) which enables to specify constraints for other languages of the family, such as class diagrams (CD) and object diagrams (OD). In Figure 8.13, for example, the class diagram `Library` defines a class `Book` which itself defines the field `edition` of type `int`. The OCL model constrains the `edition` field of `Book` to be at least `1`, i.e., the first edition (line 5). For this, it imports all classes of the class diagram (line 1) and uses its class `Book` as the invariant's context (line 4). Moreover, the invariant defines the variable `b` of the imported `Book` type and employs it to access the `edition` field of `Book` (line 5). In summary, the OCL model refers to symbols (or names) defined locally (e.g., `b`) and also refers to (imported) symbols defined in the CD model (e.g., `Book`). Having access to imported symbols, the OCL model obtains further information, such as the field `edition`.

The example shows that the composition of the CD and OCL models is solely conducted *based on names*, as mentioned above. That is, the OCL model only knows about the *names* defined in the class diagram. These names, however, can be defined in any other model (e.g., an object diagram) that adheres to this naming. Consequently, the OCL symbol table can be implemented without (statically) referring to the CD symbol table (case $L_2 \xrightarrow{knows\ not} L_1$).

Figure 8.14 highlights the essence of the model composition via (an excerpt of) the emerging scope graph. The left subgraph represents the CD model while the right subgraph represents parts of the OCL model. Please note that (the artifact scope of) `LibraryConstraint` *privately imports* (cf. Section 3.10) the `Book` class, not the whole class diagram `Library` itself, resulting from the import statement `Library.*`. The scope graph also highlights where the reference `Book` ultimately resolves to, i.e., to

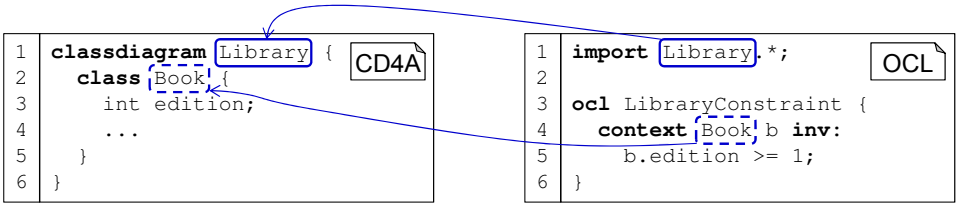


Figure 8.13: Exemplary models for aggregation of CD and OCL.

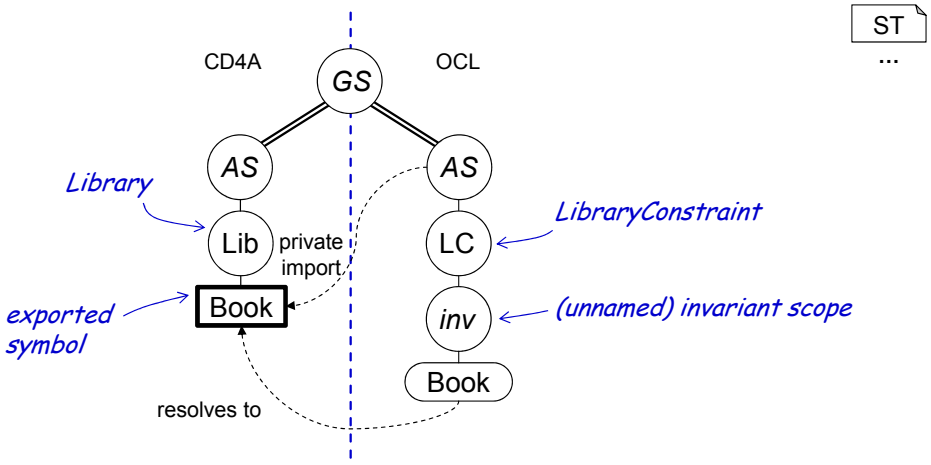


Figure 8.14: Emerging scope graph of the models in Figure 8.13. The (sub-)graphs of the models remain separated.

the same-named class defined in the CD. To this end, it has to be specified to which symbol kind the reference refers to. If the OCL language explicitly knows about the CD language (case $L_2 \xrightarrow{knows} L_1$), the respective kind (i.e., `CDTypeKind`) can be directly utilized in the reference. Consequently, the complete (generic) resolution mechanism as particularized in Chapter 6 can be reused (cf. *CRQ1*), allowing for models of these languages to be composed without further translations.

However, to decrease coupling between languages and that way facilitate reuse, the languages should be kept independent of each other (case $L_2 \xrightarrow{knows\ not} L_1$) and composed afterwards. Thus, the OCL symbol table infrastructure must provide symbols for OCL types (e.g., `OCLTypeSymbol`). OCL, though, does not specify types itself but only refers to types (of other languages). As a result, `OCLTypeSymbol` is never instantiated but rather serves as some kind of placeholder. This topic is discussed in Section 8.5.

8.3.2 Cross-Language Inter-Model Resolution

Similar to language embedding, several problems occur when aggregating the (independent) languages CD and OCL:

- Starting the resolution via `resolve("Book", OCLTypeSymbol.KIND)` excludes CD types (i.e., `CDTypeSymbol`).
- Finding the respective CD type is not sufficient since it is unknown in the OCL language.

- Unlike language embedding, language aggregation concerns several artifacts. Hence, searching for an OCL type (that should finally lead to a CD type) must also include (re-)loading of CD models as well as name qualification. This issue also exists in case $L_2 \xrightarrow{knows} L_1$ where OCL explicitly knows about CD.

As described in Section 8.2.2, SMI solves the problems mentioned in the first two items above via adapted resolving filters. For the OCL and CD example this means that a resolving filter is required which (i) translates requests like `resolve("Book", OCLTypeSymbol.KIND)` to `resolve("Book", CDTypeSymbol.KIND)` and (ii) translates the resolved `CDTypeSymbol` to a `OCLTypeSymbol` via an adapter.

The third issue, i.e., name qualification and (re-)loading of CD models, concerns the model loading process discussed in Section 6.9. In short, model loading for a single language is conducted during the top-down inter-model resolution process where `GlobalScope` delegates to the language's model loader (e.g., `CDModelLoader`) as well as model name calculator (e.g., `CDModelNameCalculator`). The latter calculates possible model names depending on the symbol's kind. Hence, invoking `calculateModelNames("Library.Book", OCLTypeSymbol.KIND)` on an instance of `CDModelNameCalculator` returns an empty collection since the OCL kind is unknown in CD. Instead, it must be translated to `calculateModelNames("Library.Book", CDTypeSymbol.KIND)`. SMI realizes this by calculating model names for every *possible symbol kind* determined by the registered resolving filters. The following demonstrates this process for resolving the `Book` class referenced in the OCL model (cf. Figure 8.13). It is assumed that the CD model is not loaded yet, and that the resolving filters `OCLTypeFilter` and `CDTypeFilter` as well as the adapted resolving filter `CD2OCLTypeFilter` are registered in the language family configuration (cf. Section 8.3.4). The resolution starts via `resolve("Book", OCLTypeSymbol.KIND)` in the OCL invariant's scope and proceeds as follows:

1. First, the (cross-language) intra-model resolution process described in Section 8.2.2 is conducted until it reaches `LibraryConstraint`'s artifact scope.
2. Then, the artifact scope determines (potential) qualified names for `Book` (i.e., it conducts name qualification), namely `Book` and `Library.Book` (resulting from the import statement `Library.*`).
3. Next, the global scope tries to resolve any of these names. For this, it determines all symbol kinds that need to be considered. These are:
   - Each (sub-)kind of `OCLTypeKind` for which a respective *non-adapting* resolving filter exists, i.e., the filter's target kind matches. In the current example, this only holds true for `OCLTypeFilter`, hence, `OCLTypeKind` is a possible kind.

- Each *source kind* of the registered *adapting* resolving filters whose target kind is kind of `OCLTypeKind`. In other words, all kinds that can be translated to the searched `OCLTypeKind` are considered. Consequently, `CDTypeKind` is a possible kind since it can be translated to `OCLTypeKind` via `CD2OCLTypeFilter`.

4. Based on these kinds (i.e., `OCLTypeKind` and `CDTypeKind`), the respective model name calculators determine possible model names. Taking the example of `CDModelNameCalculator`, it, among others, calculates model names via `calculateModelNames("Library.Book", OCLTypeSymbol.KIND)` and `calculateModelNames("Library.Book", CDTypeSymbol.KIND)`. While the former returns an empty collection since the OCL type is unknown within the CD language, the latter returns `"Library"` which is (potentially) the name of the class diagram that defines class `"Library.Book"`.

5. The CD language's model loader then loads the artifact `Library.cd` from the model path and attaches its scope graph to the global scope (cf. Section 6.9).

6. Finally, the class `Book` is resolved via the (cross-language) top-down intra-model resolution (cf. Section 6.5) extended with the translation mechanism introduced in Section 8.2.2.

8.3.3 Symbol Table Creator for the Composed Language

As highlighted in Figure 8.14, the symbol tables of the CD and the OCL models remain separated. This is different from language embedding where the symbol table elements of the embedded language are part of the host language's symbol table (cf. Figure 8.5). Thus, language aggregation does not require further customization or adaption for the symbol table creation. It works out-of-the-box, once the configuration is set, as described in the next section.

8.3.4 Language Aggregation Configuration

In particular, language aggregation only concerns the resolution process. The processing of the respective models—i.e., parsing and symbol table creation—remains unchanged and independent of the other languages. Hence, solely a composed configuration of the involved modeling languages is required. For this, SMI provides the interface `ModelingLanguageFamily` which is illustrated in Figure 8.15. A `ModelingLanguageFamily` consists of several `ModelingLanguages`. The class `CommonModelingLanguageFamily` provides the default implementation for the `ModelingLanguageFamily` interface.

In case the referenced language is known (case $L_2 \xrightarrow{knows} L_1$), no further configuration is required (cf. *CRQ3* and *CRQ4*). Else, additional (adapted) resolving filters must be added.

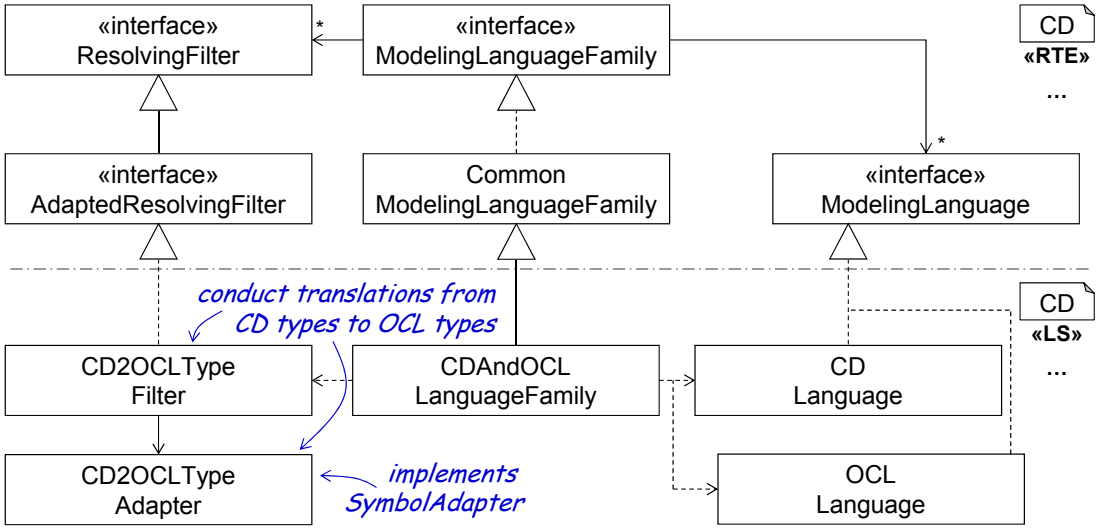


Figure 8.15: Infrastructure for language aggregation.

The bottom part of Figure 8.15 depicts the configuration for the language aggregation of the CD language and the OCL language, described in the example above. Besides the corresponding languages, `CD2OCLTypeFilter` (and with it `CD2OCLTypeAdapter`) is added to the language family to enable translations from CD types to OCL types. Since these translations concern the aggregation itself, `CD2OCLTypeFilter` is not added to a particular language but to the language family. All other components of the CD and OCL languages are fully reused from `CDLanguage` and `OCLLanguage`, respectively (cf. *CRQ2*).

As elaborated in Section 6.9.2 (on page 157), the `GlobalScope` class conducts the model loading process and already considers loading of models from different languages. That means, `GlobalScope` tries to load a model for each registered language. Listing 8.16 demonstrates how `GlobalScope` is configured for a language family.

The `GlobalScope` instance (line 5, Listing 8.16) solely requires the language family (line 1) and the model path (line 3). Henceforth, the global scope can be used same as for single languages to resolve symbols as well as load models of any registered language. Except for the configuration, nothing changes from a language user's viewpoint (cf. *CRQ5*). For example, `scope.resolve("Library.Book", OCLTypeSymbol.KIND)` returns a symbol of type `OCLTypeSymbol`, which really is an instance of `CD2OCLTypeAdapter` that adapts a `CDTypeSymbol`.

```
1 ModelingLanguageFamily family = new CDAndOCLLanguageFamily();
2
3 ModelPath modelPath = new ModelPath(...);
4
5 Scope globalScope = new GlobalScope(modelPath, family);
```

Java «HC»

Listing 8.16: Exemplary configuration of `GlobalScope` with a language family.

8.3.5 Discussion and Related Work

Language aggregation implies references between models of heterogeneous languages, and therefore, always concerns the symbol tables of the involved languages. In contrast to language embedding and language inheritance (cf. Section 8.4) the AST structure is never affected since references only occur as names.

Same as language embedding, language aggregation fully reuses the generic resolution mechanism (cf. *CRQ1*), thus, yields the same advantages, i.e., consistency between a symbol and its spanned scope, enabling the language user to rely on the language-specific symbols without the need to consider language composition (cf. *CRQ5*). In particular, name qualification—and with it model loading—is fully reused since it is conducted based on symbol kinds (cf. Section 6.3). As discussed in Section 8.2.5, this is not the case in [Völ11] which requires the additional components `IQualifiedEntryHandler` and `IQualifierClient` to adapt the qualification process for the entries.

The configuration in this thesis is similar to [Völ11]: a language family groups the respective languages and adds translations if required. Same as for language embedding, the infrastructure in [Völ11] only conducts language aggregation if resolution is started on the generic namespaces. Please refer to Section 8.2.5 for a discussion on this.

Xtext [Bet13] enables both types of language aggregation, i.e., $L_2 \xrightarrow{knows} L_1$ and $L_2 \xrightarrow{knows\ not} L_1$. In contrast to our approach, Xtext specifies aggregation within the grammar via meta-model imports. In case of $L_2 \xrightarrow{knows} L_1$, the grammar of L_2 imports L_1's meta-model. This allows to specify references to L_1's elements via the concept introduced in Section 7.3. For example, `state=Automaton::State|FQN` refers to the `State` element of the `Automaton` language via its fully qualified name. The references ultimately refer to `EObjects` of the EMF framework [SBPM09]. The imported meta-model solely has to be registered in the setup of the importing language. Similarly, in our approach the two languages are configured in a language family. Xtext conducts language aggregation for case $L_2 \xrightarrow{knows\ not} L_1$ via language inheritance in combination with the aggregation approach for case $L_2 \xrightarrow{knows} L_1$. That means, a glue grammar G_3 is required which *extends* L_2's grammar and *imports* L_1's meta-model. G_3 then overrides the respective rule(s) of L_2 grammar and creates references to elements of L_1's meta-model, as in case $L_2 \xrightarrow{knows} L_1$ described above. Consequently, the meta-model

emerging from G_3 (i.e., the AST) is an extension of L_2's meta-model with references to L_1's meta-model. In contrast, in the current thesis the ASTs of the different languages remain separated. Only the symbols are linked together. This, however, is conducted via translations using resolving filters.

Same as Xtext and the current thesis, Spoofax [KV10] allows for language aggregation with and without explicit knowledge of the referenced language. In case $L_2 \xrightarrow{knows} L_1$, the referencing language L_2's name binding module only needs to import L_1's name binding module and specify binding rules as usual (e.g., cf. Chapter 6). This is similar to our approach, where symbols of L_2 can directly refer to symbols of L_1, which only requires the symbol table creator of L_2 to consider references to symbols of L_1. Spoofax conducts language aggregation without the knowledge of the referenced language (case $L_2 \xrightarrow{knows\ not} L_1$) similar to language embedding in this thesis. An additional adapter module for syntax definition imports the two languages and accordingly extends the rules of the referencing language L_2. Moreover, an adapter module for the name binding is required which imports the name binding module of the referenced language and specifies references to its elements for the newly added rules of the syntax definition.

Since projectional approaches such as MPS [VS10] focus on direct manipulations of the AST, language aggregation in case $L_2 \xrightarrow{knows} L_1$ is realized via direct references from concepts (i.e., AST nodes) of L_2 to concepts of L_1. In case of $L_2 \xrightarrow{knows\ not} L_1$, a glue language is required which extends L_2 and extends its concepts with references to concepts of L_1.

8.4 Language Inheritance

In language inheritance a sublanguage L_{Sub} extends a superlanguage L_{Sup}. For this, the sublanguage explicitly employs elements of the superlanguage (case $L_2 \xrightarrow{knows} L_1$), while the superlanguage remains unchanged. Language inheritance includes grammar inheritance (cf. Section 2.2.2) in order to reuse or modify the syntax, enabling incremental language development [MŽ05]. Depending on the extension, language inheritance can also include extension of the superlanguage's symbol table or the methods where symbols are collected. Figure 8.17 depicts the general concept of language inheritance.

The top part of Figure 8.17 shows the AST structure as well as the symbol table of the superlanguage L_{Sup}. The bottom part depicts the respective structures of the sublanguage L_{Sub}. L_{Sub} extends L_{Sup}'s AST nodes N_k and N_m via grammar inheritance. N_k is related to a ST element which therefore is extended as well. Certainly, extending an AST node does not necessarily require extending the respective ST element as in Figure 8.17. This rather depends on the changes introduced in the AST node, i.e., whether they constitute essential information.

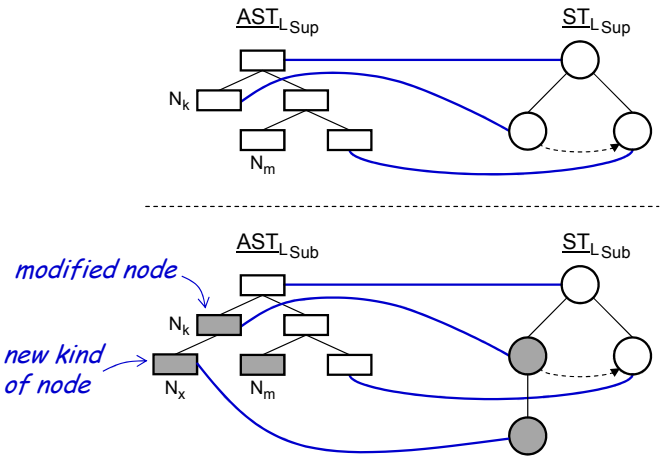


Figure 8.17: Conceptual idea of language inheritance including extension of ST elements.

Unlike N_k, the node N_m is not related to a ST element. Thus, the ST of L_{Sub} is not affected from it. Lastly, N_k in L_{Sub} introduces the new node N_x which represents an essential element of the language. Hence, it is related to a (new) ST element.

Restricting a language can lead to the removal of an AST node and with it the respective ST element. For example, a sublanguage of L_{Sup} can remove the node N_k—by overriding the respective production in the grammar—which also removes its related ST element. Alternatively, a language can be restricted via "extension of the validation phase" [EGR12], i.e., via context conditions. Following Erdweg et al. [EGR12], we consider language restriction as a special case of language inheritance (language extension in terms of [EGR12]) and do not further consider it separately.

8.4.1 Example

MontiJava [Mul15] is a language developed with MontiCore. It is an extension of Java 5 (also developed with MontiCore) both syntactically and semantically. That means:

- The syntax of Java is completely reused and extended with further constructs, such as the `singleton` keyword, which declares a class to be a singleton [GHJV95].
- It retains the same semantics for the constructs inherited from Java. Therefore, all context conditions of Java still apply for MontiJava. Additionally, further context conditions check the validity of the new constructs, for example, that an interface is not declared as singleton. To this end, MontiJava's symbol table elements subclass the respective elements of Java's symbol table.

```
1 grammar MontiJava extends Java {                                    MCG
2
3   SingletonModifier implements Modifier = "singleton";
4   ...
5 }
```

Listing 8.18: Excerpt of MontiJava's grammar which extends the grammar of Java.

The syntactic extensions are conducted via grammar inheritance (cf. Section 2.2.2), as shown in Listing 8.18. MontiJava's grammar extends Java's grammar (line 1). In order to add a new modifier, the `SingletonModifier` rule implements the `Modifier` interface rule of Java (line 3).

To provide the new information introduced in MontiJava in the symbol table, the class `MJTypeSymbol` is created and extends `JavaTypeSymbol` with the additional method `isSingleton` (cf. Figure 8.19). Analogously, `MJTypeKind` subclasses `JavaTypeKind` and enables (i) including `MJTypeSymbols` when resolving for `JavaTypeKind` and (ii) excluding `JavaTypeSymbols` when explicitly resolving for `MJTypeKind`. This is a major benefit of explicit kind hierarchies, as described in Section 3.3. In contrast, [Völ11] states symbol kinds using strings which only enable kind hierarchies via explicit translations (cf. Section 8.4.4). Since `MJTypeSymbol` extends `JavaTypeSymbol`, all algorithms (such as context conditions) based on `JavaTypeSymbol` can also be conducted on `MJTypeSymbol`.

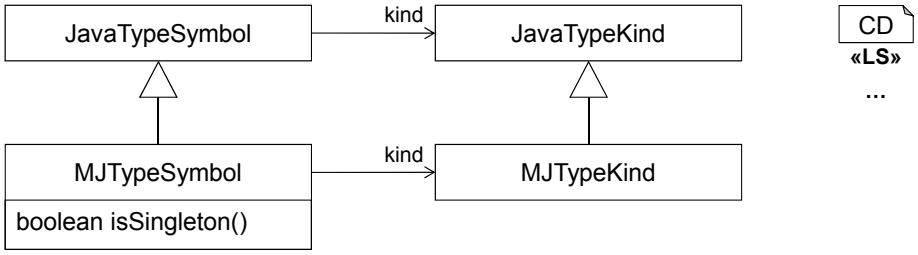


Figure 8.19: Symbol and kind hierarchy of MontiJava.

8.4.2 Symbol Table Creator for the Composed Language

To initialize the extended symbols with additional information, the symbol table creator of the superlanguage must be subclassed and customized where needed (cf. Figure 8.20). Furthermore, the language-specific visitor `MJVisitor` must be implemented to enable handling of MontiJava's specific AST nodes.

To simplify reuse, symbol table creators should follow the implementation as presented in Section 7.9. That is, the symbol table creator has to provide a dedicated Java method

for each step of the respective method introduced in Section 5.2. Also, the Abstract Factory pattern as well as the Builder pattern [GHJV95] help making symbol instantiation exchangeable (cf. Section 5.5).

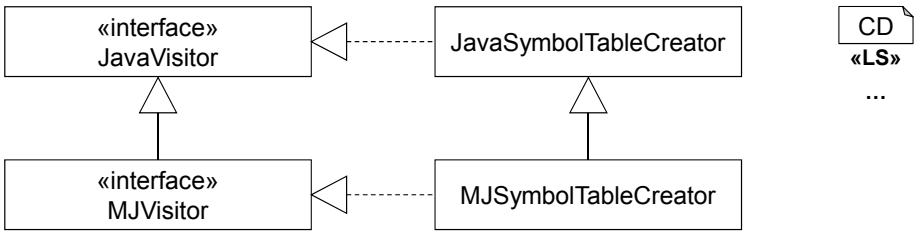


Figure 8.20: Visitor and symbol table creator of MontiJava.

8.4.3 Language Inheritance Configuration

Configuring a sublanguage, such as MontiJava, is the same as configuring any single (not composed) language (cf. Section 6.9.2). Additionally, resolving filters of the superlanguage must be registered to enable resolving of its symbols. MontiJava, for example, registers both a resolving filter for `MJTypeSymbol` and a resolving filter for `JavaTypeSymbol`.

8.4.4 Discussion and Related Work

Language inheritance does not necessarily affect the symbol table. If, for example, language L_2 extends language L_1 only with syntactic sugar, L_2 can reuse L_1's symbol table without further modifications. For instance, extending Java in order to allow the syntax `+ class` as an alternative to `public class` does not introduce new (essential) elements. The class is final and public in either case. Consequently, Java's type symbol is sufficient for this extension.

In contrast, if L_2 additionally adds essential information, it will require an extension of L_1's symbol table (for an exception to this rule see below). The CD4Analysis language, for example, is a restricted CD language of UML/P [Sch12, Rum16], which only allows for defining classes with fields. Methods do not exist. Extending CD4Analysis with method signatures in a sublanguage ExtCD4Analysis requires a new symbol `ExtCDMethodSymbol` that represents the newly introduced methods. Additionally, a new type symbol `ExtCDTypeSymbol` must extend `CDTypeSymbol` to allow storing and retrieving of method symbols, e.g., via `type.getMethods()`. Finally, ExtCD4Analysis must extend CD4Analysis' symbol table creator with handling of method symbols.

If models of L_2 *can be transformed* to respective models of L_1 obtaining the same semantics [ERKO11] (similar to refactorings [Fow99]), there is no need for extending L_1's symbol table in L_2. Certainly, this applies if the transformation(s) can be developed efficiently, e.g., via transformation languages [MCG05]. MontiJava, for example,

introduces the class `MJTypeSymbol` to represent the newly added `singleton` keyword. Alternatively, MontiJava classes can be transformed to normal Java classes which implement the Singleton pattern [GHJV95]. That way, the symbol table of Java can be completely reused. This approach has the benefit that it does not require development and maintenance of a new symbol table. However, a drawback is that the symbol table of the superlanguage does not explicitly represent the new concepts of the sublanguage. In Java, for example, the information that a class is a singleton is only specified *implicitly* (e.g., private constructor and a static `getInstance` method). Instead, MontiJava introduces a new type symbol which *explicitly* contains the new information, e.g., via the `isSingleton` method (cf. Section 8.3.1).

The hierarchical kind concept introduced in this thesis (cf. Section 3.3 and Section 4.1) simplifies language inheritance and increases development efficiency. For instance, the MontiJava type kind `MJTypeKind` extends the Java type kind `JavaTypeKind`. This enables resolving MontiJava type symbols when searching for Java type symbols, e.g., via `scope.resolve("T", JavaTypeSymbol.KIND)`. As a consequence, models of superlanguages can be fully reused in sublanguages *without* the need for translations (cf. *CRQ4*). Section 8.5 describes how this feature can be exploited to define generic symbol table infrastructures by the example of Java-like languages.

The kind hierarchy is a major difference to the infrastructure in [Völ11] where no hierarchy between symbol table entry kinds exists (cf. Section 4.1). Consequently, entries of the superlanguage must be explicitly translated to entries of the sublanguage. This not only impedes the sublanguage's configuration but also makes it more error-prone since the language engineer must be aware of the translations. Furthermore, it yields the inconsistency drawback already discussed in Sections 8.2.5 and 8.3.5. Same as [Völ11], SMI excludes symbols of the superlanguage when explicitly searching for symbols of the sublanguage. For example, resolving MontiJava type symbols, i.e., `scope.resolve("T", MJTypeSymbol.KIND)`, excludes Java type symbols.

Same as in the current thesis, language inheritance in Xtext [Bet13] includes grammar extension and only allows for the case $L_2 \xrightarrow{knows} L_1$. Also, L_2 models can be used wherever L_1 models are required since the underlying AST nodes (or meta-model) are subtyped in L_2. The same holds true for MPS [KV10] where concepts of the extended language can be extended by concepts (i.e., subconcepts) of the extending language.

In Spoofax [KV10] no additional mechanism is required in order to conduct language inheritance. The extensions are defined in additional modules (e.g., for name binding). Same as in our approach, this is only possible with explicit knowledge of the extended language (case $L_2 \xrightarrow{knows} L_1$).

8.5 Generic Symbol Table Infrastructure for Java-like Languages

Different languages can share the same or similar essential information (cf. semantic model [Fow10]). For those cases, a common symbol table infrastructure is suited which then can be specialized by concrete languages. Since Java is a prominent language, SMI provides a generic symbol table infrastructure for Java-like (or object-oriented) languages, called JST. This infrastructure is, among others, extended by MontiCore's Java language (as implemented in [Mul15]) and by CD4Analysis (restricted UML/P CD [Sch12]). Furthermore, JST serves as *reference implementation* for languages with similar concepts. The ADL MontiArc [HRR12], for example, shares some (structural) concepts similar to object-oriented languages. For instance, component inheritance is similar to class inheritance. A component can specify type arguments as well. Moreover, ports of components are akin to Java fields, having a type and a name.

Certainly, there are still major differences between MontiArc and object-oriented languages. Components, for example, can define a parameter list (analogously to methods) while classes may not. Also the semantics of a component strongly differs from that of a class in object-oriented languages. As a consequence, the corresponding symbols should not be in the same type (or kind) hierarchy. The following elucidates the core aspects and ideas of JST. Appendix D lists the complete interfaces and their implementations.

Figure 8.21 gives an overview of JST. The interface `JTypeSymbol` represents Java-like types[4] and applies the pattern *(D) Same Symbol Class for Similar Model Elements* (cf. Section 4.1.2), that is, similar model elements such as classes and interfaces are represented by the same symbol class. It is of kind `JTypeKind` and provides, among others, methods to determine which of the two types the symbol ultimately represents, i.e., `isClass` and `isInterface`. Moreover, `JTypeSymbol` gives information about the specified modifiers of the type, such as `isAbstract` and `isPublic`. Furthermore, it provides the method `getSuperClass` for retrieving the respective superclass symbol[5].

JST provides the abstract class `CommonJTypeSymbol` as default implementation for `JTypeSymbol`. This class employs type arguments in order to enable the use of specific symbols for types. For example, `CommonJTypeSymbol` as depicted in Figure 8.21, defines formal type arguments for specifying the type of type symbols (`T`) and type references (`V`). Consequently, the return type of the overridden method `getSuperClass` (inherited from `JTypeSymbol`) changes from `JTypeSymbolReference` to `V`. This enables type symbols of a concrete language to specify a specific type reference by binding `V`. In the CD4Analysis language, for example, `CDTypeSymbol` binds `V` to `CDTypeReference` to (statically) ensure that supertypes of a CD type are CD types themselves.

[4] Analogously, the interfaces `JFieldSymbol` and `JMethodSymbol` represent Java-like fields and methods, respectively (cf. Appendix D).

[5] In fact, the method wraps its return type in a `java.util.Optional` object, which, however, is omitted here for reasons of clarity.

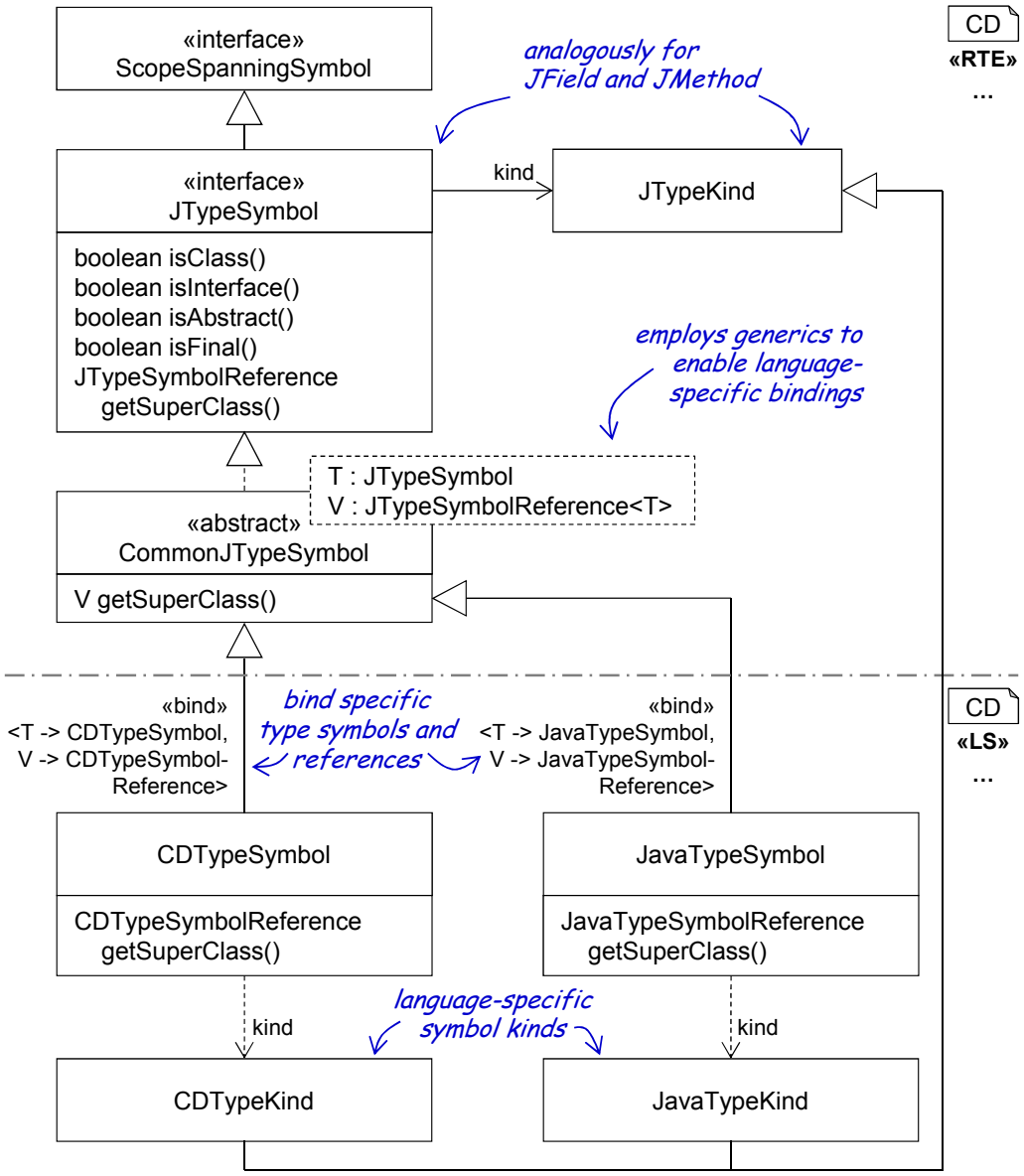


Figure 8.21: Excerpt of the Java-like symbol table infrastructure JST (cf. Appendix D).

Analogously, binding `T` to `CDTypeSymbol` ensures that only CD4Analysis type symbols are used. Furthermore, `CDTypeSymbol` employs its specific kind `CDTypeKind`, which extends `JTypeKind`. While the generic class `CommonJTypeSymbol` is difficult to read and understand due to the formal type arguments, these arguments are hidden from users of the CD4Analysis language. This facilitates the use of the language as shown below.

To enable resolution in supertypes (cf. Section 6.6), `CommonJTypeSymbol` spans a `CommonJScope` which extends `CommonScope` (not shown in Figure 8.21). As an example, if a field that is used in a class cannot be resolved within that class, `CommonJScope` tries to resolve it in the respective supertypes which includes the superclass and the implemented interfaces. If the field still cannot be resolved, the resolution continues in the class' enclosing scope (e.g., the artifact scope or an outer type's scope).

Listing 8.22 demonstrates the usage of JST by the example of CD and Java type symbols. First, an instance of `CDTypeSymbol` is defined in a variable of type `JTypeSymbol` (line 2) which then is employed to retrieve a reference to the supertype's symbol (line 3). The reference is stored in a variable of type `JTypeReference` (line 3). Similarly, a `JavaTypeSymbol` instance (line 6) and a reference to its superclass (line 7) are stored in variables of type `JTypeSymbol` and `JTypeReference`, respectively.

As it can be seen in Listing 8.22, omitting type arguments in the `JTypeSymbol` interface simplifies usage and improves readability of the code. Constructs such as `JTypeSymbol<JTypeReference> type` are not required to explicitly state that the reference is of type `JTypeReference`. In order to still enable use of specific symbols, as demonstrated in lines 10–11 of Listing 8.22, the default class `CommonJTypeSymbol` defines type arguments which are bound by `CDTypeSymbol` (and `JavaTypeSymbol`). Hence, the method `getSuperClass` can be used in a specific (line 11) or generic (line 3) way. In particular, the former enables to access language-specific information, among others, to check well-formedness and conduct code generation. The latter allows to employ JST as common denominator for languages referring to types, and hence, enables composition with languages that define those types (e.g., Java and C#). Figure 8.23 demonstrates the general idea.

Any client code (such as a symbol or context condition) can access the symbols defined in JST (since it is part of SMI). Furthermore, a language's symbol table infrastructure can extend JST and add language-specific information. A language referring to symbols

Java «HC»

```
1 // generic use of CD symbols
2 JTypeSymbol a = new CDTypeSymbol("A");
3 JTypeReference aSuper = a.getSuperClass();
4
5 // generic use of Java symbols
6 JTypeSymbol b = new JavaTypeSymbol("B");
7 JTypeReference bSuper = b.getSuperClass();
8
9 // specific use of CD symbols (analogously for Java symbols)
10 CDTypeSymbol c = new CDTypeSymbol("C");
11 CDTypeReference cSuper = c.getSuperClass();
```

Listing 8.22: Usage of JST by the example of CD and Java type symbols.

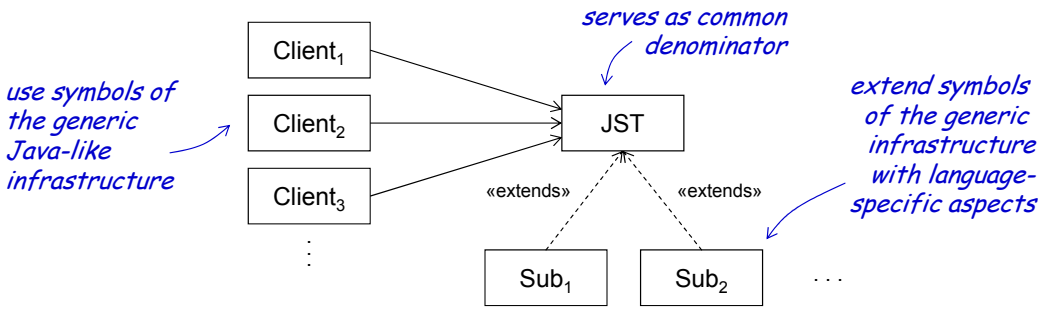


Figure 8.23: JST as common denominator. Referring to JST elements includes elements of its sublanguages.

of JST ultimately refers to symbols of a sublanguage of JST. This yields the advantage that *independent* languages (case $L_2 \xrightarrow{knows\ not} L_1$) can be composed without the need for translations (cf. *CRQ4*), i.e., neither adapted resolving filters nor symbol adapters are required (cf. *CRQ3*). Moreover, placeholder types—as `OCLTypeSymbol` of the OCL language (cf. Section 8.3)—can be omitted. Consequently, JST increases the efficiency of the language engineering process for Java-like symbol tables and reduces redundancy.

The following clarifies the benefits of JST by the example of the CD and OCL composition presented in Section 8.3. OCL does not enable type definitions. The language still introduces the placeholder classes `OCLTypeSymbol` and `OCLTypeReference` to enable type references in an OCL invariant (represented by `OCLInvariantSymbol`). To eliminate these placeholder classes, `OCLInvariantSymbol` employs JST's interface `JTypeReference`, which refers to a `JTypeSymbol`, as depicted in Figure 8.24.

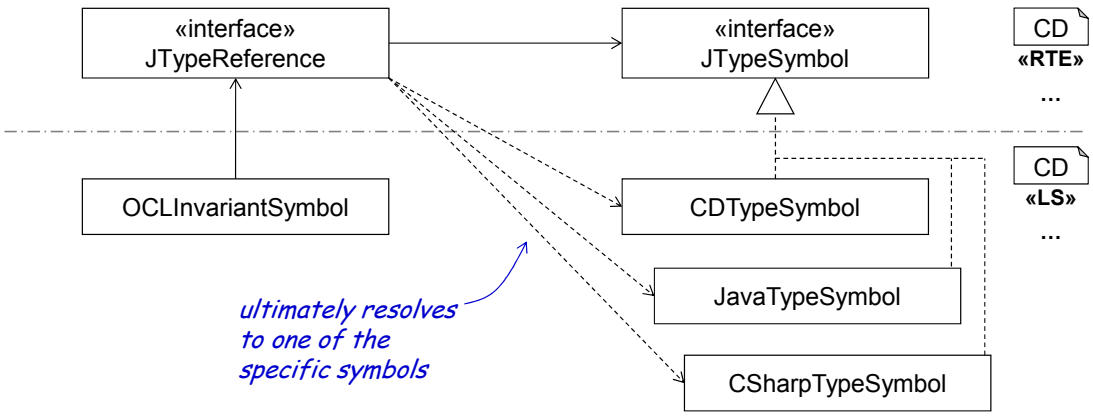


Figure 8.24: Exemplary type hierarchy of JST.

The kind hierarchy (cf. Section 4.1) together with the kind-based resolution (cf. Chapter 6) allow for utilizing any symbol that is a subkind of `JTypeSymbol`, such as `CDTypeSymbol`, `JavaTypeSymbol`, and `CSharpTypeSymbol` (cf. Figure 8.24). As a result, the composed language does not require further translations. Still, the OCL language is independent from the other languages since it only relies on the JST as part of the SMI. In contrast, composing OCL with several object-oriented languages following the approach in Section 8.3 requires a translation *for each of them*.

8.6 Non-Transitive versus Transitive Translations

The `CommonAdaptedResolvingFilter` class introduced in Section 8.2.2 only conducts direct translations. If, for example, a symbol S_1 is translated to symbol S_2 (i.e., $S_1 \rightarrow S_2$) which in turn is translated to S_3 (i.e., $S_2 \rightarrow S_3$), this does not imply that S_1 is also translated to S_3 (i.e., $S_1 \not\rightarrow S_3$). Technically, `CommonAdaptedResolvingFilter` does not delegate to adapted resolving filters. That way, transitivity is excluded.

To enable transitive adapting, SMI provides the class `TransitiveAdaptedResolvingFilter` which transitively traverses through the translation chain and conducts the respective adaptions. It is aware of cycles, i.e., $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_1$ and is left associative, i.e., $((S_1 \rightarrow S_2) \rightarrow S_3)$.

The top part of Figure 8.25 illustrates the adapter infrastructure for translating a CD type symbol to a C# type symbol that is translated to a Java type symbol, i.e, $CDType \rightarrow CSharpType \rightarrow JavaType$. For each of the translation steps $CDType \rightarrow CSharpType$ and $CSharpType \rightarrow JavaType$ a dedicated adapter class exists, i.e., `CD2CSharpTypeAdapter` and `CSharp2JavaTypeAdapter`, respectively. Please note that no adapter for $CDType \rightarrow JavaType$ exists, nor are the existing adapters statically related.

The bottom part of Figure 8.25 highlights the left associativity of the transitive adaption, i.e., $((CDType \rightarrow CSharpType) \rightarrow JavaType)$. It further shows that only one "real" symbol exists, namely `:CDTypeSymbol`. The other two instances are adapters.

Transitive translation can simplify the language composition process since not every possible composition needs to be explicitly stated. However, it yields some major issues. First, it complicates the understanding of the composition. For example, the translations $S_1 \rightarrow S_2$, $S_2 \rightarrow S_3$, $S_3 \rightarrow S_4$, and $S_4 \rightarrow S_5$ imply, among others, $S_1 \rightarrow S_3$, $S_2 \rightarrow S_4$, and $S_3 \rightarrow S_5$. Second, the transitive translations might produce unwanted results. For example, the implicit translation $S_2 \rightarrow S_4$ given above may not be wanted. To recognize whether unwanted translations emerge, the language engineer has to determine all possible (implicit) translations. Similar to multiple inheritance in object-oriented languages, transitive translation also gives rise to the so-called *diamond problem*. That is, given the translations $S_1 \rightarrow S_2$, $S_1 \rightarrow S_3$, $S_3 \rightarrow S_4$, and $S_2 \rightarrow S_4$, S_1 can be translated to S_4 via S_2 (i.e., $S_1 \rightarrow \boldsymbol{S_2} \rightarrow S_4$) or via S_3 (i.e., $S_1 \rightarrow \boldsymbol{S_3} \rightarrow S_4$). However, the resulting translated symbols are not necessarily the same which can lead to ambiguous resolutions.

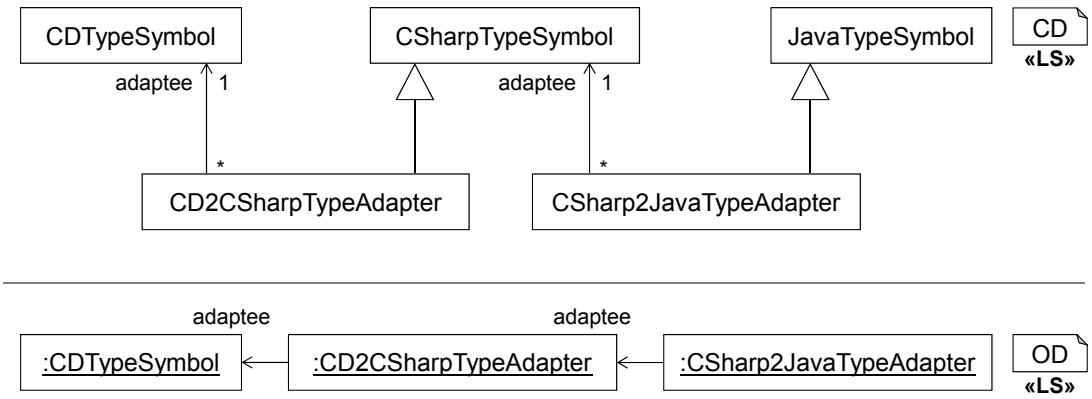


Figure 8.25: Example of transitive adaption.

8.7 Combinations of Language Compositions

Composed languages can be part of a composition themselves (cf. "extension compositions" [EGR12]). Language inheritance results in a new (composed) language (cf. Section 8.4). Hence, it can be extended as any other (single) language and also be part of a language family. The same is true for language embedding, which also results in a new composed language, but might require additional glue code (e.g., translations between symbols). Moreover, language inheritance and embedding can be combined as well, e.g., sublanguages can embed other languages. A language family (emerged from language aggregation) can be aggregated with further languages. For this, the required translations and languages only need to be registered in a (new) language family (cf. Section 8.3.4). However, a language family cannot extend or embed other languages since it consists of multiple languages.

Figure 8.26 presents a more complex example of language composition using a combination of language embedding, aggregation, and inheritance. Starting from the inside to the outside, OCL is embedded in the statechart language SC which itself is embedded in the Java language. This language composite indirectly (case $L_2 \xrightarrow{knows\ not} L_1$) refers to models of the CD4Analysis language, which has the sublanguage ExtCD4Analysis.

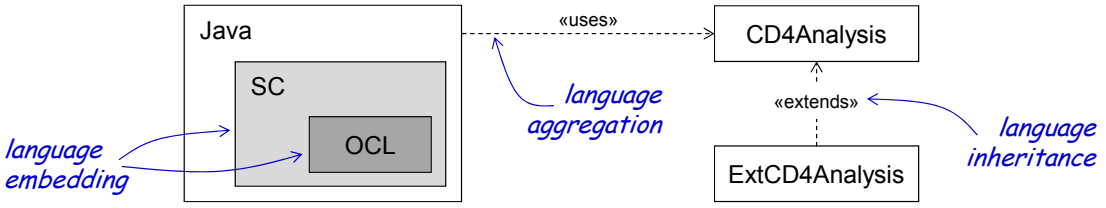


Figure 8.26: Example of a combination of language compositions.

To understand the underlying resolution process for this combination, it is useful to consider the emerging scope graph. The right part of Figure 8.27 demonstrates the scope graph for the composed model on the left.

The name `f` in OCL (line 10) resolves to the field `f` defined in the outer Java class (line 4) via a cross-language bottom-up intra-model resolution (cf. Section 8.2.2). This implies that (i) the resolution of a OCL variable is translated to the resolution of a Java field and (ii) Java fields are translated to OCL variables.

Furthermore, the OCL invariant refers to the type `T` (line 9) which ultimately resolves to the same-named class of the ExtCD4Analysis model. This is conducted by a cross-language inter-model resolution (cf. Section 8.3.2) (analogously to the previous case). The translation can be realized either by translating CD4Analysis types to OCL types or by translating ExtCD4Analysis types to OCL types. The former allows for using both CD4Analysis and ExtCD4Analysis types since ExtCD4Analysis is a sublanguage of CD4Analysis. The latter excludes CD4Analysis types and only translates ExtCD4Analysis types to OCL types (cf. Section 8.5).

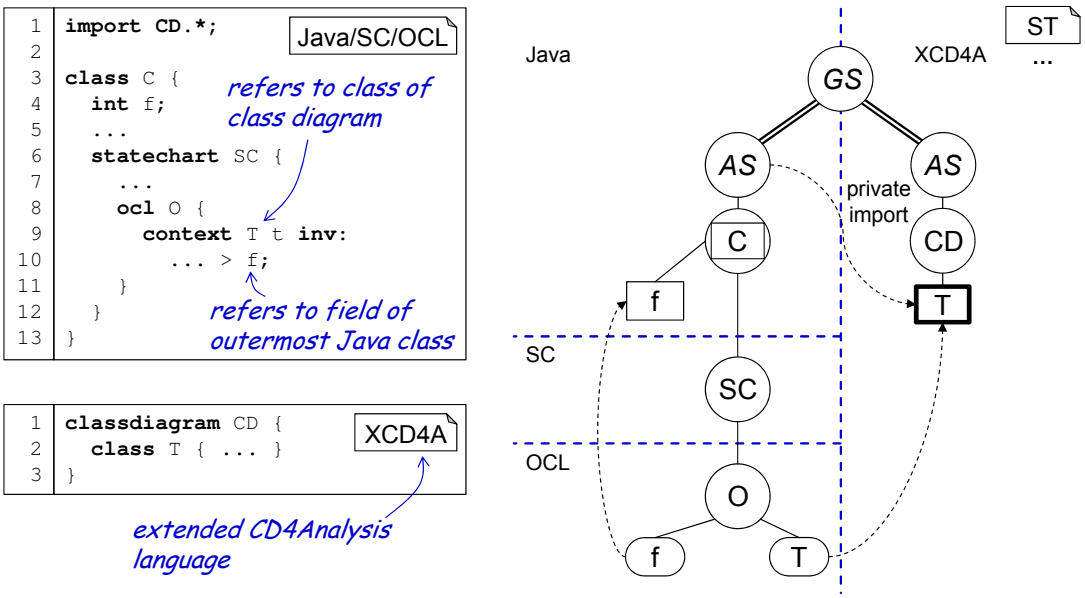


Figure 8.27: Scope graph for an exemplary model of the complex language composition in Figure 8.26.

8.8 Alternative Classifications for Language Composition

According to Erdweg et al. [EGR12], language composability "is not a property of languages themselves" but "a property of language definitions, that is, whether two definitions work together without changing them". Those definitions may only be extended in an object-oriented manner (cf. Open/Closed Principle [Mey88]) or via adding glue code [Mer13].

Mernik [Mer13] distinguishes between two groups of approaches for language composition: informal and formal. Approaches of the former group specify a language's syntax or semantics (or both) in an informal way (e.g., via a GPL), while approaches in the latter case exploit formal specifications for both syntax and semantics. The language workbench Spoofax—as part of the MetaBorg framework [BV04, BdGV06]—applies a formal approach, among others, via the meta-languages SDF, NaBL and Stratego (see discussion above). In contrast, the approach presented in this thesis only specifies the syntax formally (via MontiCore's grammar specification) but conducts the semantics programmatically, and thus, is an informal approach according to Mernik [Mer13]. Further informal approaches (as presented in [Mer13]) are Metafront [BS02, BS07], JSE [BP01], Racket [THSAC$^+$11], and SugarJ [ERKO11] (cf. Section 2.2.8) which all exploit syntax macros [Lea66] in order to translate an (extended) AST to the base language's AST during the parsing process. While this allows for language inheritance and language embedding of case $L_2 \xrightarrow{knows} L_1$, it prohibits language aggregation and embedding of independent languages (case $L_2 \xrightarrow{knows\ not} L_1$) as defined in the current thesis. This is because language aggregation requires *two independent* languages (or more) whose models can be interpreted together. The ASTs of these models *remain separated*. Hence, a translation to a base language as in syntax macros is not possible (and not required).

Moreover, Mernik [Mer13] presents *extensible compilers* which exploit an informal approach for language composition. They provide mechanisms—e.g., mixin classes and method delegation in Polyglot [NCM03] or via interception of the compilation pipeline in Helvetia [RDN09]—which allow for language inheritance as well as language embedding as in this thesis, but not for language aggregation.

Same as in MontiCore, Tatoo [CFR07], component-based LR parsing [WBGM10], and YAJCo [PFSB10] specify a language's syntax via formal grammar specifications and also employ GPLs for an informal implementation of the language's semantics. Chodarev et al. [CLPK14] utilize YAJCo to demonstrate an abstract syntax driven approach for language composition based on object-oriented as well aspect-oriented concepts. In particular, the abstract syntax is defined with Java and enriched with concrete syntax via Java's annotations. The semantics is defined via aspects (cf. [PSKM10]).

Voelter [Voe13] provides a further classification for language composition based on dependencies between the languages and syntax composition. A language L_2 depends on a language L_1 if (i) L_2 is developed with knowledge about L_1 or (ii) L_2 uses syntactic

elements of L_1. From this Voelter derives four kinds of language composition. In *referencing* models of language L_2 refer to models of L_1 *via names*. Although technically separated (e.g., in different files), models of L_2 are created with explicit knowledge about models of L_2. This corresponds to our language aggregation approach where $L_2 \xrightarrow{knows} L_1$. Similarly, in *reuse* models of the different languages remain separated. In contrast to referencing, the models are developed independently which equals language aggregation where $L_2 \xrightarrow{knows\ not} L_1$ as in this thesis. The *extension* approach denoted by [Voe13] refers to the same concepts as language inheritance in the current thesis. Furthermore, *embedding* corresponds to our embedding approach in case $L_2 \xrightarrow{knows\ not} L_1$.

Chapter 9

Summary and Future Work

In this thesis we aim at supporting the language engineer in efficiently developing infrastructures that provide essential model information in a straightforward manner to facilitate model processing and (heterogeneous) model composition. The main results of our work are briefly summarized in Section 9.1. Finally, in Section 9.2 we suggest some future work.

9.1 Summary

Explicitly providing essential information associated with a model in an additional structure has the following advantages. First, it simplifies the development of tools for model processing (such as code generators) since a model's information can be easily accessed. Second, since the essential information includes a model's interface, it facilitates composition of models from heterogeneous modeling languages. For this, the interfaces must be appropriately mapped to each other. Third, an additional structure is coupled to a language's grammar more loosely than an abstract syntax tree (AST)—at least in cases where the AST is fully (or to a large extent) generated from the grammar. As a consequence, grammar changes do not necessarily affect the additional structure which, in turn, does not require updating of tools that only employ this structure.

To increase the development efficiency of such a structure—called symbol table in this thesis—for a specific language, we (i) identified symbol table commonalities among several languages (cf. Chapter 3), (ii) developed a generic infrastructure as well as patterns and techniques based on these commonalities (cf. Chapters 4, 5, 6, and 8) and (iii) developed a generator that employs the generic infrastructure to produce parts of a language's symbol table structure (cf. Chapter 7).

In particular, Chapter 3 introduces core concepts and elements of symbol tables, namely, symbols and their kinds (which represent essential information of a model element), symbol references, scopes, scope spanning symbols, and access modifiers. Chapter 4 presents the generic infrastructure for symbol management called SMI which has been integrated into the MontiCore language workbench. SMI enables an efficient and effective development of language-specific symbol tables by providing technical realizations with reasonable defaults based on the concepts and elements introduced in Chapter 3. Furthermore, it

allows for lightweight, self-contained symbol tables which can be employed with minor configuration efforts, e.g., for static analysis. Since in MontiCore the symbol table and the AST together constitute a language's abstract syntax, SMI facilitates linking of the symbol table structure with the corresponding AST and that way simplifies the use of both structures as needed. Moreover, Chapter 4 elaborates patterns for realizing language-specific symbol table elements and discusses their impact on language composition.

Before a symbol table can be used, it first has to be built up, e.g., by traversing the AST and instantiating appropriate scopes and symbols. For this, Chapter 5 employs SMI's technical classes and provides techniques for building up language-specific symbol tables. Next, Chapter 6 demonstrates how to conduct symbol resolution requests on those built symbol tables. The resolution process is realized in a generic way, so that it not only allows for a (name-based) composition of models from a single language but also from heterogeneous languages. In Chapter 7 we exploit some of the methods and technical classes introduced in the previous chapters for generating parts of a language-specific symbol table infrastructure. Designated extension points facilitate customization of the generated code.

SMI provides ways for separating generic aspects (i.e., the scopes) from language-specific aspects (i.e., the symbols). While a symbol table user (e.g., a code generator engineer) can focus on the latter to conduct language-specific tasks in a convenient way, the former enables a non-invasive composition of symbol tables from heterogeneous languages. To enable this, in Chapter 8 we exploit the generic aspects to extend SMI, its symbol table creation process, and its symbol resolution process. Finally, we present a reference implementation for the symbol table structure of Java-like languages (called JST) which can be easily reused and extended.

Throughout this thesis, we demonstrate the applicability of SMI by the example of MontiCore's Java language [Mul15] (based on JST) and an automaton language. Moreover, SMI is employed in the following projects:

- NESTML, a modeling language family for spiking neurons [PBI+16]
- MontiJava, an extension of MontiCore's Java language (based on JST) [Mul15]
- MontiArc, an architecture description language [HRR12] (migrated from older MontiCore versions) which employs JST and constitutes a language family with MontiCore's Java language mentioned above
- Object Constraint Language (OCL) for MontiCore [Cel15]
- CD4Analysis, restricted UML/P class diagrams [Sch12, Rum16] based on JST
- UML/P object diagrams [Sch12, Rum16]

- JavaScript for MontiCore [Sie15]
- language family for robotics applications [HMSNR+15a] (migrated from older MontiCore versions)
- MontiCore's grammar language [Kra10] (migrated from older MontiCore versions)
- managing guided and unguided code generator customizations [MSNRR15]
- managing the composition of output-specific generator information [MSNRR16]
- UML activity diagrams [LN16] based on [Rei16]
- tagging language for component and connector models [MRRW16]

Many of the above listed projects have been developed simultaneously with SMI which enabled to obtain fast feedback and improve SMI in the sense of its understandability for the language engineers and users, its applicability, and its development effort.

9.2 Recommendation for Future Work

In this thesis both the AST and the symbol table constitute the abstract syntax of a language. Hence, the same model element can be simultaneously represented by an AST element as well as a symbol table element. This, however, can lead to inconsistency issues between the AST and the symbol table, for example, when modifying the AST via transformation languages [MCG05]. To keep the symbol table consistent with its corresponding AST, it must be removed and recreated every time the AST changes. Thus, a potential future investigation could be to examine how the modifications can be conducted on both structures to ensure consistency by construction.

The generative approach presented in this thesis only required a minor extension of MontiCore's grammar language, but allows to generate large parts of a language-specific symbol table. More sophisticated analyses of the extended grammar in a future work might help to generate even more parts of the symbol table (although a full-fledged generation is rarely possible, as discussed throughout this thesis). For example, determining inner symbols enables to fully generate the model name calculator (cf. Section 6.9.5). Moreover, possible extensions could be considered to increase the effectiveness of the symbol table generation. For instance, a language's configuration sets the file extension for models of that language. Adding this information, e.g., via a lightweight and simple DSL, would enable a full generation of the configuration, liberating the language engineer from handcrafted and repetitive tasks. In this thesis, we intentionally avoid complex DSLs as, e.g., in [KKWV13] since we target developers with skills in Java, and therefore, provide designated Java hook points for extension and customization of the generated code.

Furthermore, languages in MontiCore can be composed with little glue code, which so far has to be handcrafted. In cases the symbol tables of the involved languages are generated to a large extent—which might require the above mentioned extensions—parts of the glue code for the symbol table composition could be generated as well (cf. [Aßm03]).

Chapter 4 discusses many patterns for implementing language-specific symbol tables. Our generative approach employs only some of these patterns. To choose the needed patterns, several generators for the symbol table components could be implemented in a future work. The symbol table generator (cf. Section 7.4) then could be configured with the needed component generators.

Finally, code generators systematically translate models to concrete (executable) code. Several code generators can be employed to generate various parts of a software system, which then are composed to the whole system. For this, the output of a generator may depend on the output of others. For example, if a (Java code) generator produces a factory class [GHJV95] for each generated class, other generators must take this into account and generate code that employs those factories for object instantiation (instead of using the `new` operator). This knowledge about a generator's output leads to a higher coupling between generators. Therefore, it could be examined to what extent the symbol table infrastructure can help to tackle this problem. In [MSNRR16] we demonstrate a first approach which explicitly stores a generator's output-specific information in the symbol table. Other generators can access this information to generate valid code. However, the approach requires further investigations.

Bibliography

[ABB+66] Paul W. Abrahams, Jeffrey A. Barnett, Erwin Book, Donna Firth, Stanley L. Kameny, Clark Weissman, Lowell Hawkinson, Michael I. Levin, and Robert A. Saunders. The LISP 2 Programming Language and System. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 661–676, New York, NY, USA, 1966. ACM.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Comput. Netw.*, 54(15):2787–2805, October 2010.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[Aßm03] Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[Bal00] Helmut Balzert. Lehrbuch der Software-Technik. Software-Entwicklung, 2000.

[Bar98] John Barnes. *Programming in Ada 95*. International computer science series. Addison-Wesley, 1998.

[BBG+63] John W. Backus, Friedrich L. Bauer, Julien Green, Charles Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph H. Wegstein, Adriaan van Wijngaarden, and Michael Woodger. Revised Report on the Algorithm Language ALGOL 60. *Commun. ACM*, 6(1):1–17, January 1963.

[BdGV06] Martin Bravenboer, René de Groot, and Eelco Visser. *Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, chapter MetaBorg in Action: Examples of Domain-Specific Language Embedding and Assimilation Using Stratego/XT, pages 297–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280. IEEE, 2001.

[BHD+01] Mark van den Brand, Jan Heering, Arie van Deursen, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction (CC) 2001*, number 2102 in LNCS. Springer, 2001.

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[BKWA11] Christoff Bürger, Sven Karol, Christian Wende, and Uwe Aßmann. *Reference Attribute Grammars for Metamodel Semantics*, pages 22–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Blo08] Joshua Bloch. *Effective Java*. Java Series. Pearson Education, 2008.

[Boy96] John Tang Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, 1996.

[BP01] Jonthan Bachrach and Keith Playford. The Java Syntactic Extender (JSE). *SIGPLAN Not.*, 36(11):31–42, October 2001.

[Bro87] Frederick P. J. Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, April 1987.

[BS02] Claus Brabrand and Michael I. Schwartzbach. Growing Languages with Metamorphic Syntax Macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '02, pages 31–40, New York, NY, USA, 2002. ACM.

[BS07] Claus Brabrand and Michael I. Schwartzbach. The Metafront System: Safe and Extensible Parsing and Transformation. *Sci. Comput. Program.*, pages 2–20, August 2007.

[BSD99] BSD 3-Clause License. `https://opensource.org/licenses/BSD-3-Clause`, July 1999.

[BV04] Martin Bravenboer and Eelco Visser. Concrete Syntax for Objects: Domain-specific Language Embedding and Assimilation Without Restrictions. OOPSLA '04, pages 365–383, New York, NY, USA, 2004. ACM.

[Car97] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe. *Globalizing Domain-Specific Languages: International Dagstuhl Seminar, Dagstuhl Castle, Germany, October 5-10, 2014, Revised Papers*, chapter On the Globalization of Domain-Specific Languages, pages 1–6. Springer International Publishing, Cham, 2015.

[CDB+14] Benoit Combemale, Julien DeAntoni, Benoit Baudry, Robert B. France, Jean-Marc Jezequel, and Jeff Gray. Globalizing Modeling Languages. *Computer*, 47(6), 2014.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[Cel15] Abdullah Celik. Implementierung der Object Constraint Language (OCL) mit der MontiCore Language Workbench. Bachelor thesis, RWTH Aachen University, 2015. in German.

[CFR07] Julien Cervelle, Rémi Forax, and Gilles Roussel. A simple implementation of grammar libraries. *Computer Science and Information Systems*, 4(2):65–77, 2007.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, Anaheim, California, USA, 2003.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Che05] James Cheney. Toward a General Theory of Names: Binding and Scope. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning About Languages with Variable Binding*, MERLIN '05, pages 33–40, New York, NY, USA, 2005. ACM.

[CHS10] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract Delta Modeling. In *Proceedings of the Ninth International Conference on*

Generative Programming and Component Engineering, GPCE '10, pages 13–22, New York, NY, USA, 2010. ACM.

[CJKW07] Steve Cook, Gareth Jones, Stuart Kent, and Alan C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Pearson Education, 2007.

[CL83] Robert P. Cook and Thomas J. Leblanc. A Symbol Table Abstraction to Implement Languages with Explicit Scope Control. *IEEE Trans. Softw. Eng.*, 9(1):8–12, January 1983.

[CLPK14] Sergej Chodarev, Dominik Lakatoš, Jaroslav Porubän, and Ján Kollár. Abstract syntax driven approach for language composition. *Central European Journal of Computer Science*, 4(3):107–117, 2014.

[Com15] Benoit Combemale. *Towards Language-Oriented Modeling*. Accreditation to supervise research, Université de Rennes 1, December 2015.

[CvdBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. *Conceptual Model of the Globalization for Domain-Specific Languages*, pages 7–20. Springer International Publishing, Cham, 2015.

[CWW80] Lori A. Clarke, Jack C. Wileden, and Alexander L. Wolf. Nesting in Ada Programs is for the Birds. In *Proceedings of the ACM-SIGPLAN Symposium on Ada Programming Language*, SIGPLAN '80, pages 139–145, New York, NY, USA, 1980. ACM.

[DBC+15] Julien Deantoni, Cédric Brun, Benoit Caillaud, Robert B. France, Gabor Karsai, Oscar Nierstrasz, and Eugene Syriani. *Domain Globalization: Using Languages to Support Technical and Social Coordination*, pages 70–87. Springer International Publishing, Cham, 2015.

[DC90] Gerald D. P. Dueck and Gordon V. Cormack. Modular attribute grammars. *Computing Journal*, 33(2):164–172, 1990.

[DCB+15] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 25–36, New York, NY, USA, 2015. ACM.

[DK98] Arie van Deursen and Paul Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance: Research and Practice*, 10:75–92, 1998.

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM.

[ECM06] ECMA. *ECMA-334: C# Language Specification*. Fourth edition, June 2006.

[EGR12] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 7:1–7:8, New York, NY, USA, 2012. ACM.

[EH06] Torbjörn Ekman and Görel Hedin. *Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, chapter Modular Name Analysis for Java Using JastAdd, pages 422–436. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. *SIGPLAN Notices, Proceedings of the 2007 OOPSLA conference*, 42(10):1–18, 2007.

[Ekm06] Torbjörn Ekman. *Extensible compiler construction*. PhD thesis, Lund University, Sweden, 2006.

[EKR+11] Sebastian Erdweg, Lennart C.L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a Language Environment with Editor Libraries. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 167–176. ACM, 2011.

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based Syntactic Language Extensibility. OOPSLA '11, pages 391–406, New York, NY, USA, 2011. ACM.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[EvdSV+13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, WilliamR. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, GabriëlD.P. Konat, PedroJ. Molina, Martin

Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, VladA. Vergu, Eelco Visser, Kevin van der Vlist, GuidoH. Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches. In Martin Erwig, RichardF. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer International Publishing, 2013.

[EvdSV+15] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44(PA):24–47, December 2015.

[FMY92] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 223–234, New York, NY, USA, 1992. ACM.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs.* Addison-Wesley, 1999.

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.

[Fow10] Martin Fowler. *Domain-Specific Languages.* Addison-Wesley Professional, 2010.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering 2007 at ICSE.*, pages 37–54, 2007.

[GBU08] Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of concrete textual syntax mapping approaches. In *ECMDA-FA*, pages 169–184, 2008.

[GC03] Joseph D. Gradecki and Jim Cole. *Mastering Apache Velocity.* Wiley, 2003.

[GCD+12] Clément Guy, Benoît Combemale, Steven Derrien, Jim R. H. Steel, and Jean-Marc Jézéquel. *On Model Subtyping*, pages 400–415. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[GH98] Etienne Gagnon and Laurie Hendren. SableCC, An Object-Oriented Compiler Framework. In *Proceedings of TOOLS 1998*, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In Slimane Hammoudi, Luis Ferreira Pires, Philippe Desfray, and Joaquim Filipe Filipe, editors, *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pages 74–85, Angers, Loire Valley, France, February 2015. SciTePress.

[GHK+15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, volume 580 of *CCIS*, pages 112–132. Springer, 2015.

[Gho10] Debasish Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 2010.

[GJR79] Susan L. Graham, William N. Joy, and Olivier Roubine. Hashed Symbol Tables for Languages with Explicit Scope Control. *SIGPLAN Not.*, 14(8):50–57, August 1979.

[GJS+13] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[GJS+14] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Technical Report Informatik-Bericht 2006-04, Software Systems Engineering Institute, Braunschweig University of Technology, 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering*, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Monticore: a framework for the development of textual domain specific languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Proceedings of Modellierung 2006 (LNI P-82)*, 2006.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In Timothy Lethbridge, Jordi Cabot, and Alexander Egyed, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 34–43, Ottawa, Canada, 2015. ACM New York/IEEE Computer Society.

[GM10] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[GNU07] GNU Lesser General Public License. `https://www.gnu.org/licenses/lgpl-3.0.en.html`, June 2007.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, volume 6662 of *LNCS*, pages 17–32. Springer, 2011.

[GR16] Jeff Gray and Bernhard Rumpe. The evolution of model editors: browser- and cloud-based solutions. *Software & Systems Modeling*, 15(2):303–305, 2016.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 edition, 2009.

[Grö10] Hans Grönniger. *Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten.* PhD thesis, RWTH Aachen University, 2010.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems.* Shaker Verlag, 2016. (to appear).

[HBvB+94] Wolfgang Hesse, Georg Barkow, Hubert von Braun, Hans-Bernd Kittlaus, and Gert Scheschonk. Terminologie der Softwaretechnik, Ein Begriffssystem für die Analyse und Modellierung von Anwendungssystemen, Teil 1: Begriffssystematik und Grundbegriffe. *Informatik Spektrum*, 17(1):39–47, 1994.

[Hed89] Görel Hedin. An Object-Oriented Notation for Attribute Grammars. In *Proceedings of ECOOP*, pages 329–345, 1989.

[Hed00] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[Hed11] Görel Hedin. *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, chapter An Introductory Tutorial on JastAdd Attribute Grammars, pages 166–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[HF92] Paul Hudak and Joseph H. Fasel. A Gentle Introduction to Haskell. *SIGPLAN Not.*, 27(5):1–52, May 1992.

[HHJ+08] Jakob Henriksson, Florian Heidenreich, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann. Extending grammars and metamodels for reuse: the Reuseware approach. *IET Software*, 2(3):165–184, June 2008.

[HHRW15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, volume 1463 of *CEUR Workshop Proceedings*, pages 18–23, 2015.

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 114–129, Berlin, Heidelberg, 2009. Springer-Verlag.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HLMSN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In Philippe Desfray, Joaquim Filipe, Slimane Hammoudi, and Luis Ferreira Pires, editors, *Model-Driven Engineering and Software Development*, volume 580 of *Communications in Computer and Information Science*, pages 45–66. Springer International Publishing, 2015.

[HLMSN+15b] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In Slimane Hammoudi, Luis Ferreira Pires, Philippe Desfray, and Joaquim Filipe Filipe, editors, *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pages 19–31, Angers, France, February 2015. SciTePress.

[HM03] Görel Hedin and Eva Magnusson. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37 – 58, 2003.

[HMSNR+15a] Robert Heim, Pedram Mir Seyed Nazari, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Modeling Robot and World Interfaces for Reusable Tasks. In *Intelligent Robots and Systems Conference (IROS'15)*, pages 1793–1798. IEEE, 2015.

[HMSNR15b] Katrin Hölldobler, Pedram Mir Seyed Nazari, and Bernhard Rumpe. Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures. In *Proceedings of the Workshop on Domain-Specific Modeling*, DSM 2015, pages 23–30, New York, NY, USA, 2015. ACM.

[HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *European Conference on Modelling Foundations and Applications (ECMFA)*, volume 9764 of *LNCS*, pages 67–82. Springer International Publishing, 2016.

[HORM08] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic Embedding of Dsls. In *Proceedings of the 7th International*

Conference on Generative Programming and Component Engineering, GPCE '08, pages 137–148, New York, NY, USA, 2008. ACM.

[HR00] David Harel and Bernhard Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff (Part I: The Basic Stuff). Technical Report MCS00-16, Mathematics & Computer Sience, Weizmann Institute Of Sience, 2000.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[HR13] Andreas Horst and Bernhard Rumpe. Towards Compositional Domain Specific Languages. In Christophe Jacquet, Daniel Balasubramanian, Edward Jones, and Tamás Mészáros, editors, *Proceedings of the 7th Workshop on Multi-Paradigm Modeling co-located with the 16th International Conference on Model Driven Engineering Languages and Systems, MPM@MoDELS 2013, Miami, Florida, September 30, 2013.*, volume 1112 of *CEUR Workshop Proceedings*, pages 1–5. CEUR-WS.org, 2013.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, Munich, Germany, February 2011. fortiss GmbH.

[HTM] HTML Specification. `https://www.w3.org/TR/html5/`. last accessed 04/2016.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In *Fifth International Conference on Software Reuse*, pages 134–142, 1998.

[JBF11] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. *Model Driven Language Engineering with Kermeta*, pages 201–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[JCB+15] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. *Software and Systems Modeling*, 14(2):905–920, 2015.

[Kü05] Thomas Kühne. What is a Model? In *Language Engineering for Model-Driven Software Development, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl*, pages 200–0, 2005.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, volume B-108 of *Techreport*, pages 7–13. Helsinki School of Economics, October 2009.

[KKWV13] Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative Name Binding and Scope Rules. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer Berlin Heidelberg, 2013.

[Kle07] Anneke Kleppe. A Language Description is More than a Metamodel. In *Proceedings of Fourth International Workshop on Software Language Engineering, 1 Oct 2007, Nashville, USA.*, 2007.

[KLK+15] Dierk König, Guillaume Laforge, Paul King, Cédric Champeau, Hamlet D'Arcy, Erik Pragt, and John Skeet. *Groovy in Action*. Manning Publications Company, 2015.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*, ECOOP '97, pages 220–242. Springer Verlag, 1997.

[KLR96] Steven Kelly, Kalle Lyytinen, and Matti Rossi. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *CAiSE*, pages 1–21, London, UK, 1996.

[Knu68] Donald F. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 12:127–145, 1968.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988.

[KR05] Vinay Kulkarni and Sreedhar Reddy. Model-driven development of enterprise applications. In *UML Modeling Languages and Applications*, pages 118–128. Springer, 2005.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, 2010.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006*, pages 150–158, Finland, 2006. University of Jyväskylä.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling 2007*, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In G. Engels, B. Opdyke, D.C. Schmidt, and F. Weil, editors, *Proceedings of Model Driven Engineering Languages and Systems (MODELS'07)*, volume 4735 of *LNCS*, pages 286–300, Nashville, TN, USA, October 2007. Springer, Germany.

[KRV08a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Mit sprachbaukästen zur schnelleren softwareentwicklung: Domänenspezifische sprachen modular entwickeln. *Objektspektrum*, 4:42–47, 2008.

[KRV08b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular development of textual domain specific languages. In *Proceedings of Tools Europe*, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. In *International Journal on Software Tools for Technology Transfer (STTT)*, volume 12, pages 353 – 372, 2010.

[KSV09] Lennart C. L. Kats, Anthony M. Sloane, and Eelco Visser. *Compiler Construction: 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, chapter Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming, pages 142–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.

[Küh06] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[KV10] Lennart C.L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM.

[KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177, Sept 2009.

[KW94] Uwe Kastens and William M. Waite. Modularity and Reusability in Attribute Grammars. *Acta Informatica*, 31(7):601–627, 1994.

[LE13] Florian Lorenzen and Sebastian Erdweg. Modular and Automated Type-soundness Verification for Language Extensions. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 331–342, New York, NY, USA, 2013. ACM.

[Lea66] Burt M. Leavenworth. Syntax Macros and Extended Translation. *Commun. ACM*, 9(11):790–793, November 1966.

[Lee08] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008.

[LMB+01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, 2001.

[LN16] Junior Lekane Nimpa. Integration of UML Activity Diagrams @ Runtime into the Data Explorer. Master thesis, RWTH Aachen University, 2016.

[LNPR+13] Markus Look, Antonio Navarro Pérez, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In B. Combemale, J. De Antoni, and R. B. France, editors, *Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC)*, volume 1102 of *CEUR Workshop Proceedings*, Miami, Florida, USA, 2013.

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2002.

[Mas16] Mas website. `http://www.mas-wb.com`, June 2016.

[MCG05] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A Taxonomy of Model Transformations. In Jean Bezivin und Reiko Heckel, editor, *Language Engineering for Model-Driven Software Development*. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[McI68] Douglas McIlroy. Mass-Produced Software Components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[MEH09] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. Demand-driven evaluation of collection attributes. *Automated Software Engineering*, 16(2):291–322, 2009.

[Mer13] Marjan Mernik. An Object-oriented Approach to Language Compositions for Software Language Engineering. *Journal of Systems and Software*, 2013.

[Met16] MetaCase website. `http://www.metacase.com`, April 2016.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st edition, 1988.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[MLAŽ99] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Multiple Attribute Grammar Inheritance. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications,*

WAGA'99, pages 57–76, Amsterdam, The Netherlands, 1999. INRIA rocquencourt.

[MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, volume 1723 of *CEUR Workshop Proceedings*, pages 19–24, October 2016.

[MSNR15a] Pedram Mir Seyed Nazari and Bernhard Rumpe. Identifying Code Generation Candidates Using Software Categories. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, volume 580 of *CCIS*, pages 356–372. Springer, 2015.

[MSNR15b] Pedram Mir Seyed Nazari and Bernhard Rumpe. Using Software Categories for the Development of Generative Software. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 498–503. SciTePress, 2015.

[MSNRR15] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table. In *Proceedings of the Workshop on Domain-Specific Modeling*, DSM 2015, pages 37–42, New York, NY, USA, 2015. ACM.

[MSNRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, volume 254 of *LNI*, pages 133–140, 2016.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML–Revised*. MIT Press, Cambridge, MA, USA, 1997.

[Mul15] David Thien Phuc Mularski. Conception and Implementation of Monti-Java. Master thesis, RWTH Aachen University, 2015.

[MŽ05] Marjan Mernik and Viljem Žumer. Incremental programming language development. *Computer Languages, Systems & Structures*, 31(1):1–16, 2005.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proc. of International Conference on Compiler Construction (CC) 2003*, number 2622 in LNCS. Springer, 2003.

[Nes13] Stefan Nessel. Entwicklung einer Sprache für JavaScript / TypeScript. Diploma thesis, RWTH Aachen University, 2013. in German.

[NN14] Bao-Loc Nguyen Ngo. Automatische Kategorisierung von Source Code. Master thesis, RWTH Aachen University, 2014. in German.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, volume 1118 of *CEUR Workshop Proceedings*, pages 15–24, 2013.

[NTVW15] Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer Berlin Heidelberg, 2015.

[OMG03] Object Management Group. MDA Guide - Version 1.0.1 , 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[OMG13] Object Management Group. Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF) - Version 1.0.1, 2013. `http://www.omg.org/spec/ALF/1.0.1/PDF/`.

[OMG14] Object Management Group. Object Constraint Language, 2014. `http://www.omg.org/spec/OCL/2.4/PDF/`.

[OMG15a] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification - Version 1.2, 2015. `http://www.omg.org/spec/QVT/1.2/PDF/`.

[OMG15b] Object Management Group. OMG Meta Object Facility (MOF) Core Specification - Version 2.5, 2015. `http://www.omg.org/spec/MOF/2.5/PDF/`.

[OMG15c] Object Management Group. OMG Unified Modeling Language - Version 2.5, 2015. `http://www.omg.org/spec/UML/2.5/PDF/`.

[OMG15d] Object Management Group. XML Metadata Interchange (XMI) Specification - Version 2.5.1, 2015. `http://www.omg.org/spec/XMI/2.5.1/PDF/`.

[Ora13] Oracle. Oracle Database SQLJ Developer's Guide, 11g Release 2 (11.2), 2013.

[Par71] David L. Parnas. Information Distribution Aspects of Design Methodology. In *IFIP Congress (1)*, pages 339–344, Ljubljana, Yugoslavia, 1971.

[Par72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[Par10] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-specific and General Programming Languages*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*, volume 254 of *LNI*, pages 93–108. Bonner Köllen Verlag, 2016.

[PFSB10] Jaroslav Porubän, Michal Forgáč, Miroslav Sabo, and Marek Běhálek. Annotation based parser generator. *Computer Science and Information Systems*, 7(2):291–307, 2010.

[PP08] Michael Pfeiffer and Josef Pichler. A Comparison of Tool Support for Textual Domain-Specific Languages. In *Workshop on Domain-Specific Modeling*, pages 1–7, 2008.

[PQ95] Terence Parr and Russell Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Journal of Software Practice and Experience*, 25(7):789–810, July 1995.

[Pre94] Wolfgang Pree. Meta Patterns - A Means For Capturing the Essentials of Reusable Object-Oriented Design. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, ECOOP '94, pages 150–162, London, UK, UK, 1994. Springer-Verlag.

[Pre95a] Wolfgang Pree. *Design Patterns for Object-oriented Software Development*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[Pre95b] Wolfgang Pree. Hot-spot-driven framework development. In *Summer School on Reusable Architectures in Object-Oriented software Development*, pages 123–127. ACM, 1995.

[PSKM10] Jaroslav Porubän, Miroslav Sabo, Ján Kollár, and Marjan Mernik. Abstract Syntax Driven Language Development: Defining Language Semantics Through Aspects. In *Proceedings of the International Workshop on Formalization of Modeling Languages*, FML '10, pages 2:1–2:5, New York, NY, USA, 2010. ACM.

[Rai05] Chris Raistrick. Applying MDA and UML in the Development of a Healthcare System. In *UML Modeling Languages and Applications*, pages 203–218. Springer, 2005.

[RDN09] Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. Language Boxes: Bending the Host Language with Modular Language Changes. In *Software Language Engineering: Second International Conference, SLE 2009*, volume 5969 of *LNCS*, pages 274–293, Denver, United States, October 2009.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, 2016.

[Ric05] Solmi Riccardo. *Whole Platform, Ph.D. Thesis*. PhD thesis, University of Bologna, 2005.

[RR13] Dirk Reiß and Bernhard Rumpe. *Using Lightweight Activity Diagrams for Modeling and Generation of Web Information Systems*, pages 61–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, volume 215 of *LNI*, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Multi-Platform Generative Development of Component & Connector Systems using Model and Code Libraries. In *1st International Workshop on Model-Driven Engineering for Component-Based Systems (ModComp 2014)*, volume 1281 of *CEUR Workshop Proceedings*, pages 26 – 35, Valencia, Spain, September 2014.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring.* Springer Berlin, 2te edition, Juni 2012.

[Rum13] Bernhard Rumpe. Towards Model and Language Composition. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, GlobalDSL '13, pages 4–7, New York, NY, USA, 2013. ACM.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods.* Springer International, July 2016.

[SBHWP16] Adolfo Sánchez-Barbudo Herrera, Edward D. Willink, and Richard F. Paige. *A Domain Specific Transformation Language to Bridge Concrete and Abstract Syntax*, pages 3–18. Springer International Publishing, 2016.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009.

[SCC06] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional Software. In *OOPSLA*, pages 451–464, 2006.

[Sch06] Douglas C. Schmidt. Model-Driven Engineering. *Computer*, 39(2):25–31, February 2006.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P.* Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[Sco09] Michael L. Scott. *Programming Language Pragmatics.* Elsevier Science, 2009.

[Seb08] Robert W. Sebesta. *Concepts of Programming Languages.* Pearson Addison Wesley, 2008.

[Sei03] Ed Seidewitz. What models mean. *Software, IEEE*, 20(5):26–32, Sept 2003.

[Sel03] Bran Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25, Sept 2003.

[Sel06] Bran Selic. Model-Driven Development: Its Essence and Opportunities. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 313–319, Washington, DC, USA, April 2006. IEEE Computer Society.

[Sie15] Simon Tobias Siewert. Implementierung der Sprache JavaScript mit der MontiCore Language Workbench. Bachelor thesis, RWTH Aachen University, 2015. in German.

[SJ07] Jim Steel and Jean-Marc Jézéquel. On model typing. *Software & Systems Modeling*, 6(4):401–413, 2007.

[SKV10] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. A Pure Object-Oriented Embedding of Attribute Grammars. *Electronic Notes in Theoretical Computer Science*, 253(7):205 – 219, 2010. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[SQL11] International Organization for Standardization. ISO/IEC 9075-14:2011, 2011.

[SSA14] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. DeltaEcore—A Model-Based Delta Language Generation Framework. In *Modellierung*, pages 81–96, 2014.

[Sta73] Herbert Stachowiak. Allgemeine Modelltheorie, 1973.

[Str00] Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, April 2000.

[SV06] Thomas Stahl and Markus Voelter. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[SVM+13] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Van Mierlo, and Huseyin Ergin. AToMPM: A Web-based Modeling Environment. In *MODELS'13: Invited Talks, Demos, Posters, and ACM SRC*, volume 1115, Miami, 2013. CEUR-WS.org.

[THSAC+11] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages As Libraries. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM.

[VANT+15a] Hendrik Van Antwerpen, Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. Technical report, Delft University of Technology, Software Engineering Research Group, 2015.

[VANT+15b] Hendrik Van Antwerpen, Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. Language-Independent Type-Dependent Name Resolution. Technical report, Delft University of Technology, Software Engineering Research Group, 2015.

[VBD+13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[VC15] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

[vdSCL14] Tijs van der Storm, William R. Cook, and Alex Loh. The design and implementation of Object Grammars. *Science of Computer Programming*, pages 460–487, 2014.

[Vis97] Eelco Visser. *Syntax Definition for Language Prototyping, Ph.D. Thesis*. PhD thesis, University of Amsterdam, 1997.

[Vis01] Eelco Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies - System Description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01), volume 2051 of Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, 2001.

[Vli98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.

[Voe13] Markus Voelter. *Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*, chapter Language and IDE Modularization and Composition with MPS, pages 383–430. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. 2011. Shaker Verlag, 2011.

[VOSC14] Edoardo Vacchi, Diego Mathias Olivares, Albert Shaqiri, and Walter Cazzola. Neverlang 2: A Framework for Modular Language Implementation. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 29–32, New York, NY, USA, 2014. ACM.

[VRKS13] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):339–390, 2013.

[VS10] Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with mps. *Software Language Engineering, SLE*, page 16, 2010.

[VSB+13] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. Wiley, 2013.

[VSK89] Harald H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher order attribute grammars. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 131–145, New York, NY, USA, 1989. ACM.

[VWH09] Eric Van Wyk and Mats Per Erik Heimdahl. Flexibility in Modeling Languages and Tools: A Call to Arms. *Journal on Software Tools for Technology Transfer*, 11(3):203–215, 2009.

[Wat04] David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons, 2004.

[WBGK10] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39 – 54, 2010. Special Issue on {ETAPS} 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

[WBGM10] Xiaoqing Wu, Barrett R. Bryant, Jeff Gray, and Marjan Mernik. Component-based LR Parsing. *Computer Languages, Systems & Structures*, 36(1):16–33, April 2010.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying Current Research in Object-oriented Design. *Commun. ACM*, 33(9):104–124, September 1990.

[WCW88] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. A model of visibility control. *Software Engineering, IEEE Transactions on*, 14(4):512–520, Apr 1988.

[WGM89] André Weinand, Erich Gamma, and Rudolf Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2), 1989.

[Wir71] Niklaus Wirth. The Programming Language Pascal. *Acta Inf.*, 1(1):35–63, March 1971.

[WMBK02] Eric van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In *Proceedings of the 11th International Conference on Compiler Construction 2002*, pages 128–142, London, UK, 2002. Springer-Verlag.

[WWM+07] Thomas Weigert, Frank Weil, Kevin Marth, Paul Baker, Clive Jervis, Paul Dietz, Yexuan Gui, Aswin Van Den Berg, Kim Fleer, David Nelson, et al. Experiences in deploying model-driven engineering. In *SDL 2007: Design for Dependable Systems*, pages 35–53. Springer, 2007.

[www16a] Apache Maven website. `https://maven.apache.org/`, April 2016.

[www16b] Eclipse Modeling Framework (EMF). `http://www.eclipse.org/modeling/emf/`, June 2016.

[www16c] Eclipse website. `http://www.eclipse.org`, May 2016.

[www16d] FreeMarker website. `http://freemarker.org/`, June 2016.

[www16e] GitHub website. `https://github.com/`, April 2016.

[www16f] Groovy Language website. `http://groovy-lang.org/`, April 2016.

[www16g] IntelliJ IDEA website. `https://www.jetbrains.com/idea/`, May 2016.

[www16h] Meta Programming System (MPS) website. `http://www.jetbrains.com/mps/`, February 2016.

[www16i] MontiArc website. `http://www.monticore.de/languages/montiarc/`, May 2016.

[www16j] MontiCore GitHub website. `https://github.com/MontiCore`, April 2016.

[www16k] Rascal website http://www.rascal-mpl.org/, April 2016.

[www16l] SugarJ website http://www.sugarj.org/, April 2016.

[www16m] The Eclipse Graphical Modeling Project. http://www.eclipse.org/modeling/gmp/, April 2016.

[www16n] The Spoofax Language Workbench website. http://www.spoofax.org, February 2016.

[ZH11] Dmitrijs Zaparanuks and Matthias Hauswirth. *Vision Paper: The Essence of Structural Models*, pages 470–479. Springer Berlin Heidelberg, 2011.

Appendix A

Index of Abbreviations

AS	Artifact Scope
ADL	Architecture Description Language
API	Application Programming Interface
AST	Abstract Syntax Tree
CD	Class Diagram
DSEL	Domain-Specific Embedded Language
DSL	Domain-Specific Language
EBNF	Extended Backus-Naur Form
GS	Global Scope
GPL	General-Purpose Language
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JST	Java-like Symbol Table Infrastructure
LHS	Left-Hand Side
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
NaBL	Name Binding Language
OCL	Object Constraint Language
OD	Object Diagram
OMG	Object Management Group
RHS	Right-Hand Side
SC	Statechart
SMI	Symbol Management Infrastructure
ST	Symbol Table
STE	Symbol Table Entry
UML	Unified Modeling Language

Appendix B

Diagram and Listing Tags

Stereotype	Description
«GEN»	Generated language-specific element (i.e., «LS»).
«HC»	Handcoded element.
«LS»	Language-specific element which is either generated (i.e., «GEN») or handcoded (i.e., «HC»).
«METH»	Method which describes (partly automated) workflows.
«MODEL»	A model is presented (e.g., if Java is used as an action language).
«RTE»	Generic element that is part of MontiCore's (or SMI's) runtime environment.

Table B.1: Explanation of the used stereotypes within listings and tags.

Tag	Description
AD	Activity Diagram
CD	Class Diagram[1]
CD4A	Restricted Class Diagram for Analysis
CpD	Component Diagram
Groovy	Groovy Script
Java	Java Source Code
MCG	MontiCore Grammar
SC	Statechart Diagram
SD	Sequence Diagram
ST	Symbol Table / Scope Graph
OCL	Object Constraint Language
OD	Object Diagram
...	Tags an incomplete diagram.

Table B.2: Explanation of the used tags in listings and figures.

[1] CD members are public if not otherwise stated.

Appendix C

Groovy Script for Specifying Model Processing Workflows

Groovy

```
1 package de.monticore
2
3 // basic setup and initialization; enabling of reporting
4 initGlobals(_configuration)
5
6 // ############################################################
7 // the first pass processes all input grammars up to
8 // transformation to CD and storage of the resulting CD to disk
9 while (grammarIterator.hasNext()) {
10   input = grammarIterator.next()
11   if (force || !isUpToDate(input)) {
12     cleanUp(input)
13
14     // M2: parse grammar
15     astGrammar = parseGrammar(input)
16
17     if (astGrammar.isPresent()) {
18       astGrammar = astGrammar.get()
19
20       startReportingFor(astGrammar, input)
21
22       // populate symbol table
23       astGrammar = createSymbolsFromAST(symbolTable, astGrammar)
24
25       // execute context conditions
26       runGrammarCoCos(astGrammar, symbolTable)
27
28       // transform grammar AST into Class Diagram AST
29       astClassDiagram = transformAstGrammarToAstCd
30         (glex, astGrammar, symbolTable, handcodedPath)
31
32
```

```
33        astClassDiagramWithST =
34          createSymbolsFromAST(symbolTable, astClassDiagram)
35
36        // write Class Diagram AST to the CD-file (*.cd)
37        storeInCdFile(astClassDiagramWithST, out)
38
39        // generate parser
40        generateParser(astGrammar, symbolTable, handcodedPath, out)
41        generateParserWrappers
42          (astGrammar, symbolTable, handcodedPath, out)
43
44        // store result of the first pass
45        storeCDForGrammar(astGrammar, astClassDiagramWithST)
46      }
47    }
48 }
49 // ############################################################
50
51 // ############################################################
52 // the second pass
53 // do the rest which requires already created CDs of possibly
54 // local super grammars etc.
55 for (astGrammar in getParsedGrammars()) {
56   // make sure to use the right report manager again
57   reportingFor(astGrammar)
58
59   astClassDiagram = getCDOfParsedGrammar(astGrammar)
60
61   // decorate Class Diagram AST
62   decorateCd(glex, astClassDiagram, symbolTable, handcodedPath)
63
64   // generate symbol table
65   generateSymbolTable(astGrammar, symbolTable,
66     astClassDiagram, out, handcodedPath)
67
68   // generate AST classes
69   generate(glex, symbolTable, astClassDiagram, out, templatePath)
70
71   info("Grammar " + astGrammar.getName()
72     + " processed successfully!")
73
74   // flush reporting
75   flushReporting(astGrammar)
76 }
```

Listing C.1: Groovy script for processing MontiCore grammars.

Appendix D

Technical Realization of the Java-like Symbol Table Infrastructure JST

D.1 Technical Realization of Java-like Type Symbols

Java «RTE»

```java
package de.monticore.symboltable.types;

import java.util.List;
import java.util.Optional;
import de.monticore.symboltable.ScopeSpanningSymbol;
import de.monticore.symboltable.types.references.JTypeReference;

/**
 * @author Pedram Mir Seyed Nazari
 */
public interface JTypeSymbol
      extends TypeSymbol, ScopeSpanningSymbol {

  JTypeSymbolKind KIND = new JTypeSymbolKind();

  boolean isGeneric();

  List<? extends JTypeSymbol> getFormalTypeParameters();

  Optional<? extends JTypeReference<? extends JTypeSymbol>>
    getSuperClass();

  List<? extends JTypeReference<? extends JTypeSymbol>>
    getInterfaces();

  List<? extends JTypeReference<? extends JTypeSymbol>>
    getSuperTypes();
  List<? extends JFieldSymbol> getFields();
```

```
29
30   Optional<? extends JFieldSymbol> getField(String name);
31
32   List<? extends JMethodSymbol> getMethods();
33
34   Optional<? extends JMethodSymbol> getMethod(String name);
35
36   List<? extends JMethodSymbol> getConstructors();
37
38   List<? extends JTypeSymbol> getInnerTypes();
39
40   Optional<? extends JTypeSymbol> getInnerType(String name);
41
42   /**
43    * @return true, if type is an abstract class or an interface
44    */
45   boolean isAbstract();
46
47   boolean isFinal();
48
49   boolean isInterface();
50
51   boolean isEnum();
52
53   boolean isClass();
54
55   /**
56    * @return true, if this type is an inner type, such as an
57    *         inner interface or inner class
58    */
59   boolean isInnerType();
60
61   boolean isPrivate();
62
63   boolean isProtected();
64
65   boolean isPublic();
66
67   /**
68    * @return true, if this type itself is a formal type parameter.
69    */
70   boolean isFormalTypeParameter();
71 }
```

Listing D.1: The `JTypeSymbol` interface of JST.

Java «RTE»

```
1  package de.monticore.symboltable.types;
2
3  import com.google.common.collect.ImmutableList;
4  import de.monticore.symboltable.CommonScopeSpanningSymbol;
5  import de.monticore.symboltable.MutableScope;
6  import de.monticore.symboltable.modifiers.BasicAccessModifier;
7  import de.monticore.symboltable.types.references.JTypeReference;
8  import de.se_rwth.commons.logging.Log;
9
10 import java.util.ArrayList;
11 import java.util.Collection;
12 import java.util.List;
13 import java.util.Optional;
14 import java.util.stream.Collectors;
15
16 import static com.google.common.base.Preconditions.checkArgument;
17 import static com.google.common.base.Strings.isNullOrEmpty;
18 import static de.monticore.symboltable.Symbols.
19     sortSymbolsByPosition;
20
21 /**
22  * @author Pedram Mir Seyed Nazari
23  */
24 public abstract class
25     CommonJTypeSymbol <T extends JTypeSymbol,
26                        S extends JFieldSymbol,
27                        U extends JMethodSymbol,
28                        V extends JTypeReference<T>>
29       extends CommonScopeSpanningSymbol implements JTypeSymbol {
30
31   private final JFieldSymbolKind fieldKind;
32   private final JMethodSymbolKind methodKind;
33
34   private V superClass;
35   private final List<V> interfaces = new ArrayList<>();
36
37   private boolean isAbstract = false;
38   private boolean isFinal = false;
39   private boolean isInterface = false;
40   private boolean isEnum = false;
41   private boolean isFormalTypeParameter = false;
42   // e.g., inner interface or inner class
43   private boolean isInnerType = false;
44
45
```

```
46  protected CommonJTypeSymbol(String name,
47                              JTypeSymbolKind typeKind,
48                              JFieldSymbolKind fieldKind,
49                              JMethodSymbolKind methodKind) {
50    super(name, typeKind);
51
52    this.fieldKind = fieldKind;
53    this.methodKind = methodKind;
54  }
55
56  protected CommonJTypeSymbol(String name) {
57    this(name, JTypeSymbol.KIND,
58      JFieldSymbol.KIND, JMethodSymbol.KIND);
59  }
60
61  @Override
62  protected MutableScope createSpannedScope() {
63    return new CommonJTypeScope(Optional.empty());
64  }
65
66  @Override
67  public boolean isGeneric() {
68    return !getFormalTypeParameters().isEmpty();
69  }
70
71  public void addFormalTypeParameter(T formalTypeParameter) {
72    checkArgument(formalTypeParameter.isFormalTypeParameter());
73    getMutableSpannedScope().add(formalTypeParameter);
74  }
75
76  @Override
77  public List<T> getFormalTypeParameters() {
78    final Collection<T> resolvedTypes =
79      getSpannedScope().resolveLocally(T.KIND);
80    return resolvedTypes.stream().filter(T::isFormalTypeParameter)
81      .collect(Collectors.toList());
82  }
83
84  @Override
85  public Optional<V> getSuperClass() {
86    return Optional.ofNullable(superClass);
87  }
88
89  public void setSuperClass(V superClass) {
90    this.superClass = superClass;
91  }
```

```
92
93  @Override
94  public List<V> getInterfaces() {
95    return ImmutableList.copyOf(interfaces);
96  }
97
98  public void addInterface(V superInterface) {
99    this.interfaces.add(Log.errorIfNull(superInterface));
100 }
101
102 @Override
103 public List<V> getSuperTypes() {
104   final List<V> superTypes = new ArrayList<>();
105   if (getSuperClass().isPresent()) {
106     superTypes.add(getSuperClass().get());
107   }
108   superTypes.addAll(getInterfaces());
109   return superTypes;
110 }
111
112 public void addField(S field) {
113   getMutableSpannedScope().add(Log.errorIfNull(field));
114 }
115
116 @Override
117 public List<S> getFields() {
118   return sortSymbolsByPosition(
119     getSpannedScope().resolveLocally(fieldKind));
120 }
121
122 @Override
123 public Optional<S> getField(String name) {
124   return getSpannedScope()
125     .resolveLocally(name, fieldKind);
126 }
127
128 public void addMethod(U method) {
129   checkArgument(!method.isConstructor());
130   getMutableSpannedScope().add(method);
131 }
132
133 @Override
134 public List<U> getMethods() {
135   final Collection<U> resolvedMethods =
136     getSpannedScope().resolveLocally(methodKind);
137
```

```
138     final List<U> methods =
139       sortSymbolsByPosition(resolvedMethods.stream()
140         .filter(method -> !method.isConstructor())
141           .collect(Collectors.toList()));
142
143     return methods;
144   }
145
146   @Override
147   public Optional<U> getMethod(String name) {
148     Optional<U> method = getSpannedScope()
149       .resolveLocally(name, methodKind);
150
151     if (method.isPresent() && !method.get().isConstructor()) {
152       return method;
153     }
154
155     return Optional.empty();
156   }
157
158   public void addConstructor(U constructor) {
159     checkArgument(constructor.isConstructor());
160     getMutableSpannedScope().add(constructor);
161   }
162
163   @Override
164   public List<U> getConstructors() {
165     final Collection<U> resolvedMethods =
166       getSpannedScope().resolveLocally(methodKind);
167
168     final List<U> constructors =
169       sortSymbolsByPosition(resolvedMethods.stream()
170         .filter(U::isConstructor).collect(Collectors.toList()));
171
172     return constructors;
173   }
174
175   public void addInnerType(T innerType) {
176     getMutableSpannedScope().add(innerType);
177   }
178
179   @Override
180   public List<T> getInnerTypes() {
181     return sortSymbolsByPosition(
182       getSpannedScope().resolveLocally(getKind()));
183   }
```

```
184
185   @Override
186   public Optional<T> getInnerType(String name) {
187     return getSpannedScope()
188       .resolveLocally(name, getKind());
189   }
190
191   @Override
192   public boolean isClass() {
193     return !isInterface() && !isEnum();
194   }
195
196   // additional getter and setter methods
197 }
```

Listing D.2: The `CommonJTypeSymbol` class of JST provides default implementations for `JTypeSymbol`.

D.2 Technical Realization of Java-like Field Symbols

```java
package de.monticore.symboltable.types;

import de.monticore.symboltable.Symbol;
import de.monticore.symboltable.types.references.JTypeReference;

/**
 * @author Pedram Mir Seyed Nazari
 */
public interface JFieldSymbol extends Symbol {

  JFieldSymbolKind KIND = new JFieldSymbolKind();

  JTypeReference<? extends JTypeSymbol> getType();

  boolean isStatic();

  boolean isFinal();

  boolean isParameter();

  boolean isPrivate();

  boolean isProtected();

  boolean isPublic();
}
```

Listing D.3: The `JFieldSymbol` interface of JST.

```java
package de.monticore.symboltable.types;

import de.monticore.symboltable.CommonSymbol;
import de.monticore.symboltable.modifiers.BasicAccessModifier;
import de.monticore.symboltable.types.references.JTypeReference;

/**
 * @author Pedram Mir Seyed Nazari
 */
public abstract class CommonJFieldSymbol
  <T extends JTypeReference<? extends JTypeSymbol>>
      extends CommonSymbol implements JFieldSymbol {

```

```
14  private T type;
15
16  private boolean isFinal;
17  private boolean isStatic;
18  private boolean isParameter = false;
19
20  public CommonJFieldSymbol(String name,
21      JFieldSymbolKind kind, T type) {
22    super(name, kind);
23    this.type = type;
24  }
25
26  @Override
27  public T getType() {
28    return type;
29  }
30
31  public void setType(T type) {
32    this.type = type;
33  }
34
35  @Override
36  public boolean isStatic() {
37    return isStatic;
38  }
39
40  @Override
41  public boolean isFinal() {
42    return isFinal;
43  }
44
45  public void setParameter(boolean isParameter) {
46    this.isParameter = isParameter;
47  }
48
49  @Override
50  public boolean isParameter() {
51    return isParameter;
52  }
53
54  public void setPrivate() {
55    setAccessModifier(BasicAccessModifier.PRIVATE);
56  }
57
58  public void setProtected() {
59    setAccessModifier(BasicAccessModifier.PROTECTED);
```

```
60    }
61
62    public void setPublic() {
63      setAccessModifier(BasicAccessModifier.PUBLIC);
64    }
65
66    @Override
67    public boolean isPrivate() {
68      return getAccessModifier()
69        .equals(BasicAccessModifier.PRIVATE);
70    }
71
72    @Override
73    public boolean isProtected() {
74      return getAccessModifier()
75        .equals(BasicAccessModifier.PROTECTED);
76    }
77
78    @Override
79    public boolean isPublic() {
80      return getAccessModifier()
81        .equals(BasicAccessModifier.PUBLIC);
82    }
83
84    // additional setter methods
85  }
```

Listing D.4: The `CommonJFieldSymbol` class of JST provides default implementations for `JFieldSymbol`.

D.3 Technical Realization of Java-like Method Symbols

Java «RTE»

```java
package de.monticore.symboltable.types;

import java.util.List;

import de.monticore.symboltable.ScopeSpanningSymbol;
import de.monticore.symboltable.types.references.JTypeReference;

/**
 * @author Pedram Mir Seyed Nazari
 */
public interface JMethodSymbol extends ScopeSpanningSymbol {

  JMethodSymbolKind KIND = new JMethodSymbolKind();

  JTypeReference<? extends JTypeSymbol> getReturnType();

  List<? extends JFieldSymbol> getParameters();

  List<? extends JTypeSymbol> getFormalTypeParameters();

  List<? extends JTypeReference<? extends JTypeSymbol>>
    getExceptions();

  boolean isAbstract();

  boolean isStatic();

  boolean isConstructor();

  boolean isFinal();

  boolean isEllipsisParameterMethod();

  boolean isPrivate();

  boolean isProtected();

  boolean isPublic();
}
```

Listing D.5: The `JMethodSymbol` interface of JST.

Java «RTE»

```java
package de.monticore.symboltable.types;

import static com.google.common.base.Preconditions.checkArgument;
import static de.monticore.symboltable.Symbols.
    sortSymbolsByPosition;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.stream.Collectors;

import com.google.common.collect.ImmutableList;
import de.monticore.symboltable.CommonScopeSpanningSymbol;
import de.monticore.symboltable.modifiers.BasicAccessModifier;
import de.monticore.symboltable.types.references.JTypeReference;
import de.se_rwth.commons.logging.Log;

/**
 * @author Pedram Mir Seyed Nazari
 */
public abstract class
    CommonJMethodSymbol<U extends JTypeSymbol,
                        T extends JTypeReference<? extends U>,
                        S extends JFieldSymbol>
      extends CommonScopeSpanningSymbol implements JMethodSymbol {

  private boolean isAbstract = false;
  private boolean isStatic = false;
  private boolean isFinal = false;
  private boolean isConstructor = false;
  private boolean isEllipsisParameterMethod = false;

  private T returnType;
  private List<T> exceptions = new ArrayList<>();

  public CommonJMethodSymbol(String name,
      JMethodSymbolKind kind) {
    super(name, kind);
  }

  @Override
  public T getReturnType() {
    return returnType;
  }

```

```
46  public void setReturnType(T type) {
47    this.returnType = Log.errorIfNull(type);
48  }
49
50  @Override
51  public List<S> getParameters() {
52    final Collection<S> resolvedFields =
53      getSpannedScope().resolveLocally(S.KIND);
54
55    final List<S> parameters =
56      sortSymbolsByPosition(resolvedFields.stream()
57        .filter(S::isParameter).collect(Collectors.toList()));
58
59    return parameters;
60  }
61
62  public void addParameter(S paramType) {
63    checkArgument(paramType.isParameter());
64    getMutableSpannedScope().add(paramType);
65  }
66
67  public void addFormalTypeParameter(U formalTypeParameter) {
68    checkArgument(formalTypeParameter.isFormalTypeParameter());
69    getMutableSpannedScope().add(formalTypeParameter);
70  }
71
72  @Override
73  public List<U> getFormalTypeParameters() {
74    final Collection<U> resolvedTypes =
75      getSpannedScope().resolveLocally(U.KIND);
76
77    return resolvedTypes.stream()
78      .filter(U::isFormalTypeParameter)
79        .collect(Collectors.toList());
80  }
81
82  @Override
83  public List<T> getExceptions() {
84    return ImmutableList.copyOf(exceptions);
85  }
86
87  public void setExceptions(List<T> exceptions) {
88    this.exceptions = exceptions;
89  }
90
91
```

```
92   public void addException(T exception) {
93     this.exceptions.add(exception);
94   }
95
96   public void setAbstract(boolean isAbstract) {
97     this.isAbstract = isAbstract;
98   }
99
100  @Override
101  public boolean isAbstract() {
102    return isAbstract;
103  }
104
105  public void setStatic(boolean isStatic) {
106    this.isStatic = isStatic;
107  }
108
109  @Override
110  public boolean isStatic() {
111    return isStatic;
112  }
113
114  public void setConstructor(boolean isConstructor) {
115    this.isConstructor = isConstructor;
116  }
117
118  @Override
119  public boolean isConstructor() {
120    return isConstructor;
121  }
122
123  @Override
124  public boolean isFinal() {
125    return isFinal;
126  }
127
128  public void setFinal(boolean isFinal) {
129    this.isFinal = isFinal;
130  }
131
132  @Override
133  public boolean isEllipsisParameterMethod() {
134    return isEllipsisParameterMethod;
135  }
136
137
```

```
138   public void setEllipsisParameterMethod (
139       boolean isEllipsisParameterMethod) {
140     this.isEllipsisParameterMethod = isEllipsisParameterMethod;
141   }
142
143   public void setPrivate() {
144     setAccessModifier(BasicAccessModifier.PRIVATE);
145   }
146
147   public void setProtected() {
148     setAccessModifier(BasicAccessModifier.PROTECTED);
149   }
150
151   public void setPublic() {
152     setAccessModifier(BasicAccessModifier.PUBLIC);
153   }
154
155   @Override
156   public boolean isPrivate() {
157     return getAccessModifier()
158       .equals(BasicAccessModifier.PRIVATE);
159   }
160
161   @Override
162   public boolean isProtected() {
163     return getAccessModifier()
164       .equals(BasicAccessModifier.PROTECTED);
165   }
166
167   @Override
168   public boolean isPublic() {
169     return getAccessModifier()
170       .equals(BasicAccessModifier.PUBLIC);
171   }
172
173 }
```

Listing D.6: The `CommonJMethodSymbol` class of JST provides default implementations for `JMethodSymbol`.

Appendix E

Curriculum Vitae

Family Name	Mir Seyed Nazari
First Name	Pedram
Birthday	12.09.1983
Birthplace	Tehran / Iran
Nationality	German
Since 01/2017	Senior Technical Consultant Senacor Technologies AG Bonn
12/2011–11/2016	RWTH Aachen University: Ph.D. studies in Software Engineering
04/2008–09/2011	RWTH Aachen University: Computer Science studies Diploma in Computer Science
10/2004–03/2008	Ruhr University Bochum: Applied Computer Science studies Bachelor in Applied Computer Science
06/2004	Luisen-Gymnasium Düsseldorf: German Abitur

List of Figures

Listings

List of Tables

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve reliability of adaprivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design

guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe

variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

References

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.