[JRS24] N. Jansen, B. Rumpe, D. Schmalzing:
A Synopsis on Import Statements in Modeling Languages.
In: Workshop on Modeling Language Engineering, MODELS Companion '24:
International Conference on Model Driven Engineering Languages and Systems (MLE),
pp. 1161–1169, Association for Computing Machinery (ACM), Oct. 2024.

A Synopsis on Import Statements in Modeling Languages

Nico Jansen
jansen@se-rwth.de
Software Engineering
RWTH Aachen University
Germany

Bernhard Rumpe
rumpe@se-rwth.de
Software Engineering
RWTH Aachen University
Germany

David Schmalzing
schmalzing@se-rwth.de
Software Engineering
RWTH Aachen University
Germany

Abstract

The composition of different source artifacts is a fundamental mechanism for modularization and reuse in many software languages and essential for separating distinct concerns and building reuseable libraries. This form of composition relies on importing foreign artifacts, i.e., including code, models, and concepts. Although importing artifacts is a concept found frequently in software languages, especially programming languages, there are different ways of doing so. Current realizations are highly verbose, and there is no uniform application concept for imports within the modeling discipline. In this paper, we analyze different approaches to realizing imports for software languages and identify their advantages and drawbacks. Based on this information, we elaborate on the beneficial features of different modeling circumstances. This collection supports future import realizations for languages by guiding the selection of the correct technique concerning different goals and corresponding features.

CCS Concepts

• **Software and its engineering → Abstraction, modeling and modularity**.

Keywords

Language Engineering, Import Statements, Model Composition, Cross-Referencing

ACM Reference Format:

Nico Jansen, Bernhard Rumpe, and David Schmalzing. 2024. A Synopsis on Import Statements in Modeling Languages. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3652620.3688347

1 Introduction

Model-driven engineering (MDE) [52] has become increasingly popular over the last few decades. In overall systems engineering, as well as specific application domains such as avionics [34], automotive [5], and robotics [64], models are increasingly established and are gradually becoming the central development documents that steer the development process. Models are abstract representations of an entity or a state of affairs used for a specific purpose [53], such as analysis or synthesizing executable program artifacts [63]. Typically, they correspond to modeling languages that define the set of valid sentences and provide corresponding tooling for processing, such as parsers, well-formedness rules, code generators, or interpreters [8]. The discipline for developing such languages, irrespective of whether for classical programming or modeling, is called software language engineering (SLE) [36]. New techniques and procedures are conceived in this domain to efficiently design, implement, maintain, evolve, and compose languages.

Modeling languages are continuously evolving [17], and their development process has matured with the establishment of sophisticated language workbenches [15]. Likewise, their application in a wide variety of disciplines is growing. Current modeling languages, such as UML [46], SysML [47], or its successor, SysML v2 [49], follow the path paved by programming languages over the past decades and support the separation of concerns [31, 39] by providing means to split functionalities into distinct artifacts. However, since these individual artifacts relate to each other and interact, they must be able to refer to each other. In programming, cross-referencing was developed for this purpose, which allows elements of other artifacts to be accessed externally. Additionally, import statements allow for pointing at these external entities and thus achieve inter-artifact operability [19].

In contemporary programming languages (e.g., Java [2], Python [58], C# [28], C++ [56], etc.) and generally in developing large software systems, the partitioning of source code into separate artifacts is widespread. Separation of source code is common in object-oriented programming [55], where individual instantiable classes can or must be outsourced to dedicated artifacts. Furthermore, the resulting modularization enables the establishment of libraries, fostering software component reusability and increasing development efficiency and quality [21]. Prominent modeling languages or language families, such as UML [46] and SysML [47], also adopted this notion, providing concepts for incorporating external elements. SysML v2 [49], in particular, is prominent in this respect, as it not only specifies the concept of imports and model libraries but also provides a suitable implementation within the pilot implementation[1]. Such concepts for imports and cross-referencing of artifacts are also occasionally employed by other modeling languages (e.g., architecture description languages [18, 26]). Additionally, some language workbenches such as Xtext [16], MPS [61], or MontiCore [29] also come with language components and techniques that facilitate realizing imports.

Despite numerous implementations in programming languages and different adaptations in modeling, these realizations are significantly different conceptually and in terms of their technical details.

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3688347

[1] https://github.com/Systems-Modeling/SysML-v2-Pilot-Implementation

Thus, these individual solutions have different implications, advantages, and disadvantages. Therefore, for many modeling languages, engineers must often develop import statement concepts largely from scratch again and evaluate the various scenarios and application use cases. This paper aims to analyze existing approaches and facilitate future decisions on the type of imports. We explore different import practices from programming and modeling, classify them, and highlight their benefits and weaknesses. Thus, the main contributions of this paper are:

- An analysis of the different features of import statements based on applied languages from programming and modeling
- An assessment of the implications with (dis-)advantages of the various mechanisms with respect to different modeling situations

The remainder is structured as follows. In section 2, we introduce the background knowledge on the topic. In section 3, we consider related work and the state of the art concerning import statements. In section 4, we elaborate on the different features and facets of imports and provide an overview of how these mechanisms function and what they are precisely referencing. We evaluate the extracted features in section 5 and indicate useful application scenarios. Finally, in section 6, we discuss our findings, and section 7 concludes.

2 Background

Our work is mainly focused on import statements for modeling languages. Therefore, we introduce the corresponding basic terminology and concepts on cross-referencing, importing, and basics on modeling languages.

2.1 Modeling Languages

For MDE to be applied successfully, models must be unambiguous and machine-processable. To this end, they correspond to formalized modeling languages that specify the set of valid models and their explicit meaning. These languages are defined by [10, 11, 25]: (1) an abstract syntax that describes the internal structure of the models, stripped of syntactic sugar, (2) a concrete syntax depicting the actual representation of the models, e.g., textual, graphical, etc., (3) a semantic domain that represents the universe into which the models are mapped [27], and (4) a semantic mapping, giving the models their actual meaning by linking the syntactic constructs into the semantic domain.

Such languages can be defined in various ways. Typical approaches are via (often) context-free grammars (CFGs) [37], metamodeling [12, 54], or projectional meta-editing [7]. CFGs enable an integrated definition of concrete and abstract syntax. Metamodels only describe the abstract structure of a language. The concrete (often graphical) representation is created in a second effort via an editor [41]. Projectional approaches are slightly different by comparison. They enable direct manipulation of the internal structure via projections (textual or graphical). However, these projectional views must also be defined in advance at the language level.

Regardless of their technological background, all models are internally represented in the form of an abstract syntax tree (AST). It only represents the model's essential structure purged from its concrete presentation. On the AST, further operations, such as well-formedness checks [10], model transformations [1, 33], analyses, or code synthesis via a generator [4, 22, 32, 50] can then be executed.

2.2 Cross-Referencing and Symbol Tables

While input models are parsed internally to ASTs for further processing, this alone is generally insufficient. By their inherent nature, ASTs are limited in cross-referencing across this tree structure. However, since references are necessary in software languages (e.g., type usages or function calls), it must also be possible to resolve such symbolic links across the AST [6]. For this purpose, the notion of the symbol table [30] has been developed.

Symbol tables are a concept that originated from compiler construction [3], which has been generally adapted for the field of SLE. Symbols are identified by a unique name [29]. They elevate the AST into a graph-like structure, which allows efficient cross-referencing and fast navigation. In general, symbols represent the essential named constituents of a model, which can be referenced and accessed internally and externally. In this context, they can also be augmented with additional information. Overall, symbols are hierarchically organized in scopes, which span namespaces, managing external visibility and, thus, accessibility [62]. Modern language workbenches support resolving via implicit or explicit realization of symbol tables.

2.3 Import Statements

While symbol management infrastructures enable referencing within an AST, they also allow the establishment of inter-model connections. Symbol tables enable resolving elements across artifact boundaries using their unique, fully qualified names. In programming languages like Java, this mechanism links modular functionalities and separates the classes in object-oriented development. A symbol resolution algorithm enables the resolution of model elements of a single language and may also support resolution across language boundaries, thus enabling language aggregation [29].

To facilitate such referencing (or, in some cases, to make it possible in the first place), import statements [19] have been developed. These allow pointing at external artifacts or their elements and thus making them accessible within the namespace of the own context. This has the effect that referenced symbols in the model or program code do not have to be addressed as fully qualified but only need their simple name, which is then completed via the import statement when resolving. Import statements, therefore, act as access pointers to external sources. Over the past decades, various realizations of import statements have emerged for a wide range of software languages. In this paper, we examine their different facets and features (cf. section 4).

3 State of the Art and Related Work

Various software (especially programming) languages provide import mechanisms with different realizations and implications. The literature also proposes several approaches and discusses these. Furthermore, current language workbenches provide different importing defaults for developing modeling languages.

With the advent of modularization in programming, Felleisen and Friedman provide a first overview of import (and export) statements in the literature [19]. They argue on the subject of programming languages that modularization also makes sense for smaller languages. This is a trend that we can also observe in the current evolution of modeling languages. They use the Scheme programming language [13] to demonstrate how modules can be incorporated into a software language and how their cross-referencing is implemented. For this purpose, the term module is first defined as a set of declarations of functions, constants, and variables, which are (partially) exposed to the outside and thus made available for external use [19]. Elements of such a module are referenced via a unique module identifier and subsequently imported. The paper also discusses three different expansion stages of import statements. These are adapted directly from evaluation strategies for expressions [14]:

Import-by-value addresses the problem that several imported elements from different modules can have the same name, leading to clashes in the program code. To tackle this problem, the aliasing concept is introduced by referencing elements via their export name in the statement but then renaming them for further use. This prevents name clashes, as functions, variables, etc., with the same name can be used via their aliases.

The first mechanism has the disadvantage that the import statement is evaluated directly during the evaluation. This results in early binding of individual components and a strict processing sequence (e.g., compilation) of the distinct artifacts. The import-by-name extension provides a lazy reload strategy. This means that import statements are only resolved when the corresponding name of the referenced module is used in the program code. As a result, imported but unused modules are not loaded unnecessarily.

Import-by-need is a further extension of the evaluation strategy, which ensures that modules are not only loaded lazily but explicitly only once. If an import statement is used multiple times, the imported module is only loaded once and then cached for further use, which results in a considerable increase in efficiency.

The efforts around Import2vec [57] also deal with the topic of import statements in various programming languages. Their primary focus is more on using modified natural language processing [9] techniques to detect semantic similarities in libraries. This is achieved by extracting imports and analyzing their related occurrences in the program code. Nonetheless, achieving this requires addressing the structure and composition of the various import statements. Overall, six languages are analyzed: Java [2], JavaScript [38], Python [58], Ruby [20], PHP [40], and C# [28] throughout distributed repositories. While these statements have many structural similarities, custom characteristics still need to be addressed appropriately. These include wildcards, renaming, and diverging import targets (e.g., individual elements vs. complete source files).

In the work on metasemantic protocols for modeling [51], Ed Seidewitz elaborates on the prerequisites for self-extension mechanisms for modeling languages. This means that languages are equipped with a generic mechanism that allows modelers to add new constructs at the model level. Consequently, language expansions are no longer exclusively dependent on the language developers, thus shifting from the M2 to the M1 level regarding the Meta Object Facility [43]. Such a metasemantic protocol is the basis for every self-extension mechanism of a language and the foundation for establishing referenceable model libraries. The work builds on well-established foundations for programming languages [35]. These concepts are transferred on the concrete example of the early phase of SysML v2 [49] and its underlying base language, Kernel Modeling Language [48], thus following the trend of adapting programming concepts to the world of modeling languages [60]. Establishing such a metasemantic protocol requires concrete and abstract syntax for its usage in the modeling language and formal semantics to determine the meaning of these syntactic constructs [51]. This results in different referencing options, often expressed throughout import statements. Their different (abstract) syntactic constructs, as well as implicit and explicit purposes, are explained in our work.

4 Different Aspects of Import Statements

Over the past decades, numerous different and occasionally contradictory approaches have emerged in the design of import statements. We list these aspects and give examples of their application in real-world modeling and programming languages. We specify the precise focus of this work before explaining the different design features in detail.

4.1 Scope and Terminology

We restrict our analysis accordingly to avoid confusion in which elements are precisely examined and the choice of terms used. By import statement, we refer to the syntactic construct for referencing language-specific artifacts (i.e., files) or elements within them. The elaboration concerns the different constituents of such statements, how and where they are realized, as well as their implicit meaning and potential impact.

The scope explicitly does not include referencing fully packaged model libraries. We associate the downloading and embedding of these with dependency management, which is usually realized by build management tools such as Maven [59] or Gradle [44]. An import statement in the context of this contribution deals exclusively with the facets of the mechanism enabling access to elements of one namespace in another. Exchange formats, such as XMI [45], are also not within the scope of this paper. Although they facilitate the exchange between different model artifacts, their (de-) serialization is independent of the actual referencing and, thus, of the import statements themselves.

For import statements, we will also show examples of concrete syntactic sentences from real-world languages. These are solely for comprehension purposes and are no prescription of concrete syntax per se. For instance, the occurrence of a keyword to indicate an import statement is typical. For our analysis, however, only its existence is relevant and not whether the exact name of the keyword is `import`, `require`, or `use`. Furthermore, cross-language imports, which are common in language aggregation [29], are neither explicitly included nor excluded, as their basic features are the same as regular imports within a single language.

4.2 Import Statement Design Aspects

We explain the different design aspects of import statements and their various manifestations. Where applicable, we give a tangible

real-world example. We divide our analysis into three categories: intrinsic characteristics, which deal with design decisions of internal realization; explicitly provided modeling features; and compound aspects in between.

4.2.1 *Intrinsic Characteristics.*

Intrinsic characteristics on import statements are mainly concerned with how they are realized internally. This primarily affects the technical infrastructure that must be provided for a language. However, these aspects can also directly influence the modeling to a certain extent.

A1: Evaluation Strategy An inherent design decision is how and when import statements are evaluated. Naturally, they are checked for syntactic correctness and well-formedness during parsing, but this does not directly imply loading a referenced source. Targeted resources can either be loaded *instantly* or *lazily*. In the case of instant import, all referenced sources are loaded and processed immediately. With the lazy load strategy, they are only loaded when they are required for processing. Although this usually has no direct influence on the model's structure, as described in [19], it can influence the efficiency of an import mechanism. In the case of interpreted languages such as Python [58], however, this can even have a concrete influence on the result, depending on the implementation, as imported modules are evaluated during loading. If the time or sequence of this evaluation changes, this also influences the calculation result.

A2: Import Execution When executing an import, referenced elements must be loaded and integrated. However, the way in which this integration is performed has various implications for the reusability of model parts. As in software development, the different forms of reusability are also distinguished in modeling. A *complete inclusion* makes external model elements available by copying all elements directly into the own model. This method is supported, for example, in the graphical modeling tool MagicDraw [41], which merges external models into its own containment tree. Another option is a *loose coupling* of the referenced elements, which are loaded locally into the memory but technically remain separate sources. This is a typical approach in common programming languages such as Java [2].

A3: Transitivity Import statements naturally incorporate the referenced elements. However, imported elements again may require further ones to operate correctly. An important decision, therefore, lies in how these transitively required elements are incorporated. One possibility is to design the import mechanism to be not just shallow, but also *transitive*. This would automatically load the elements required in a transitive yet implicit manner. The Java programming language [2] supports this paradigm. This means that loaded elements work without additional effort, such that their elements and functionalities can be accessed, while transitive class and method definitions are not explicitly available in the current namespace. Another possibility is a completely *shallow* import, which requires the modeler to explicitly handle transitive dependencies with additional import statements. This means an increased modeling effort with greater control and flexibility at the same time. The R programming language [23], for instance, offers an option for such non-transitive imports.

4.2.2 *Explicit Features.*

Explicit features are concrete syntactic constructs in import statements that pursue a direct objective. These usually add additional modeling options and either extend the capabilities for importing or specify shortcuts for improved usability.

A4: Wildcards Import statements reference an external source that should be loaded and incorporated. This is usually done by fully qualifying the required source. Additionally, many languages support using a wildcard in the import statement to combine several imports and facilitate usability. The wildcard is an additional syntactic construct (often expressed as `*`), which leaves the exact referenced source underspecified and can, therefore, serve as a placeholder for multiple targets. For instance, the statement `import java.util.*` imports several classes and interfaces from the corresponding Java library, such as `List`, `Vector`, `Date`, and many more.

A5: Aliasing Since elements in different modules can have identical names, name clashes arise when importing several such modules. As a result, the referenced element can no longer be clearly identified. The principle of aliasing tackles this issue, which allows elements to be renamed during their import. As a result, the referenced elements with the same name from external namespaces have a distinct and, therefore, unique name in the importing namespace. Python allows such renaming by introducing the additional keyword `"as"`. For instance, through the statement `import vehicles.Automobile as Car`, the external class `Automobile` is known locally as `Car` and can be used.

A6: Visibility Different visibilities for elements are a well-established concept in both programming and modeling to configure their accessibility from the outside. This concept can also be applied to import statements. In SysML v2 [49], for instance, an import can be explicitly specified as public or private to indicate whether imported elements should also be exported again. That is, technical details or elements that are only relevant for the internal use of a module can be hidden by private imports. Visibility should not be confused with transitive imports (see **A3**), which only implicitly grant chained access to incorporated object functionalities. Public or private imports directly influence the number of externally visible elements.

4.2.3 *Compound Aspects.*

Compound properties in import statements cannot be assigned to intrinsic or explicit categories. They are based on internal design decisions, which can be enhanced by additional explicit modeling features. Vice versa, there are also (seemingly harmless) syntactic constructs that can significantly affect internal namespacing.

A7: Import Location While imports make elements of external namespaces available to their own, their respective position in the model can have some semantic and technical implications. In many programming languages, such as Java, imports are specified at a dedicated position as the beginning of an artifact, making referenced constituents available in all subscopes of the *namespace* spanned by the *artifact*. In contrast, SysML v2 also allows imports in any subscopes (e.g., attribute definitions), which enables a more flexible and *distributed import* of elements. The two approaches,

therefore, have fundamentally different meanings since, in the first case, imports are only possible at the artifact level. At the same time, flexible localization requires technical support for imports in every subscope.

A8: Import Targets Although imports fulfill the task of making external elements accessible, the exact target kind to which the statement points can vary. Thus, artifacts, language elements, or a combination of these can be referenced. A classic example of *pure artifact* imports is scripting languages such as Bash [24]. Here, only the path to another script file to be included is specified. It is then loaded, and its functions are accessible. The exact opposite of this approach comprises import statements, which point *exclusively* to the *elements* contained in the artifact, regardless of its localization. An example of this is SysML v2 [49], which uses artifact modularization exclusively for structuring purposes but does not consider artifact borders in imports. Many programming languages employ an *integration* of both approaches. For example, the qualified name of an import statement (as in Java [2]) is primarily aimed at the artifact to be incorporated, which methodically has the same name as the class, interface, etc., to be included. Only when directly referencing inner classes or static methods and constants does the import statement also directly contain the name of the referenced language elements. This means that the statement is split into two parts in such a combination. The first (main) part points to the artifact to be imported, while inner elements are specified as its suffix. Other languages, such as Python [58], introduce an additional syntactic construct in the import statement, which *separates* the module reference from the module content. The statement `from automotive import vehicles.Car` thus incorporates the `Car` class from the `automotive` component, allowing the module name and internal package structure to be treated separately.

A9: Cyclicality Several modeling artifacts can reference each other and import their respective mutual elements. This results in direct or indirect cyclic dependencies, which makes the execution order of affected artifacts indeterminable. While considered a code smell in many software engineering scenarios, such cycles can occasionally be practical. In general, there are three main options to deal with such circular dependencies and the resulting infinite reload loop. If we *prohibit cycles* (as in the Go language [42]), we avoid all negative effects but also deprive modelers of the ability to use them in a targeted manner. Alternatively, cycles can still be allowed, but manual *import guarding* is required. This implies that a modeler effectively allows modules to be imported under certain conditions. However, this guarding requires an additional syntactic construct. An example of this can be found in more traditional languages such as C++ [56]. An advanced approach to allowing such cyclic imports would be for the language tool to automatically perform *cycle extraction* in the dependency graph of all incorporated artifacts and process the corresponding ones jointly once (cf. Java [2]).

A10: Relative Imports Classical import statements specify the *absolute* (qualified) name of the target to be incorporated. However, some languages, such as Python [58], also offer the option of *relative* import statements. These do not identify the target qualified from the project root but from the referencing artifact itself. This can have advantages regarding cleaner source code if the required functionality is located in a namespace close to your own.

5 Aspect Evaluation for Modeling

Based on the gathered features for import statements, we analyze their applicability in the context of modeling languages. Finally, we summarize our results and aggregate the findings into a compact, configurable set of default import statements suitable for modeling.

5.1 Analysis and Recommendations

While different features may benefit certain domains but not others, we try to establish a general baseline for modeling. This means that features will be examined primarily with respect to fundamental modeling benefits, such as the abstraction of technical details and consistent integration into a context. Furthermore, we will also discuss the technical implications of realizing such features to provide language engineers with an overview of the implementation requirements. Table 1 aggregates our assessment results explained in the following.

A1: Evaluation Strategy The choice of evaluation strategy has minimal impact on the actual modeling experience. However, a lazy load strategy positively influences the efficiency of the reloading mechanism. Sources only need to be loaded on demand; they can be buffered for faster multiple access, and unused imports have no negative effect, as they are ignored by construction. An instant evaluation strategy leads to the front-loading of all imported sources, whether required or not. Naturally, the dynamic approach requires the resolve algorithm to reload and manage missing sources adaptively. Although this mechanism is more complex than front-loading through instant import evaluation, language workbenches can provide it generically. This means that a language developer would only have to use this functionality with regard to the designed import statements. The lazy load strategy is, therefore, clearly recommended to ensure loading efficiency even in more complex modeling languages with many larger models involved.

A2: Import Execution Regarding the type of execution of imports, loose coupling is clearly preferable to full inclusion. Integrating external model elements completely into the current model has rather the characteristics of copy-paste reuse, even if it is performed automatically by a modeling tool. In comparison, almost all modern programming languages exclusively use the principle of loose import coupling. Just imagine modern IDEs loading the complete program code of the referenced source into the current artifact upon import. This would completely pollute the code base. The same applies to modeling. Another disadvantage of complete inclusion is that it is difficult or even impossible to react to changes in a library, as you are not working on the referenced sources but on a local copy. This would destroy the agile and integrative nature of MDE. Therefore, it is essential that the import mechanisms of state-of-the-art modeling languages are based on loose coupling of artifacts.

A3: Transitivity Whether imports should be transitive or non-transitive by default depends mainly on the application domain. In highly technical areas (such as hardware-related software development), customized imports may be required to fill transitively

Table 1: The various aspects of import statements and their characteristics assessment concerning modeling.

A1: Evaluation Strategy	
instant ◔	lazy ◕
A2: Import Execution	
complete inclusion ○	loose coupling ●
A3: Transitivity	
shallow ◔	transitive ●
Explicit Features		
A4: Wildcards ◕	A5: Aliasing ◕	A6: Visibility ◔
A7: Import Location	
artifact namespace ◕	distributed import ◕
A8: Import Targets			
pure artifact ○	exclusively elements ◓	integration ◕	separator ◕
A9: Cyclicality		
prohibit ◓	import guarding ◔	cycle extraction ●
A10: Relative Imports	
absolute ●	relative ◓

Legend:
● highly recommended
◕ beneficial
◓ neutral
◔ discouraged
○ not suitable

required holes, which is easily possible with non-transitive imports. However, the overall modeling activity is rather abstracted from technical details, reducing the need for non-transitive imports in reality. Moreover, transitive imports significantly increase usability. Otherwise, a modeler would have to manually enter all transitive imports, which is a tedious and error-prone manual effort. Accordingly, transitive reloading algorithms should be preferred in modeling but are also significantly more complicated to implement. However, such an algorithm can be delivered by default in language workbenches as basic functionality in a library. Therefore, imports should operate transitively by default, and only an optional possibility of non-transitive imports should be considered for specific application domains.

A4: Wildcards The application of wildcards has been widely used in both programming and modeling over the last few years. They facilitate referencing multiple model elements with a single statement, thus keeping code and models cleaner and more manageable. However, in some cases, wildcards could lead to unwanted ambiguities in referencing, resulting in errors in the model. Nevertheless, this issue can be identified at design time via well-formedness rules, such that the modeler can be warned. In conjunction with aspect **A1**, wildcards show the importance of following a lazy load reloading strategy to ensure that only the sources actually used are reloaded for wildcards and not all those in the specified namespace automatically. Additionally, tooling support could automatically refactor wildcard imports into the respective required explicit ones. Since these general drawbacks can be compensated by standard validation rules and recommended evaluation strategies, the advantages in terms of usability outweigh. Therefore, including wildcards as a general construct is reasonable and should be supported by default.

A5: Aliasing Aliases alter the names of external entities within the importing namespace. The primary purpose of this mechanism is to avoid name clashes with duplicate names from different modules. Without the aliasing, we would otherwise have to ensure that no name duplicates exist globally in library artifacts, which is often beyond a modeler's control. Furthermore, it creates convenience functionality for assigning more suitable names during import. In general, aliasing has no real disadvantage except that, in daily modeling, it may arguably not be the most frequently used. Overall, however, they are a helpful and valuable construct that can be recommended as a default feature for imports.

A6: Visibility Visibilities in import statements help specify which incorporated elements are exported again to the outside. While modeling scenarios exist where explicit hiding or forwarding of employed elements might be helpful, the general benefit is debatable. Modeling generally abstracts away from the technical details of the underlying tooling. In conjunction with the recommended transitivity (cf. **A3**), implicit access to all necessary functionalities is generally provided anyway. Furthermore, an incorrect understanding of this feature and different modeling conventions regarding visibility during imports can lead to stress and inconsistencies. Therefore, it can be useful for selected domain-specific languages or for languages with a general universality claim (such as the general-purpose language SysML v2 [49]). In most cases, however, language engineers should refrain from providing this feature.

A7: Import Location Distributed import statements with local integration are more flexible to use but can lead to more confusing models. Furthermore, saving imports in each spanned scope must be possible to realize them technically. With imports at artifact level, this only has to happen in one central location for each file. Even if the infrastructure is of varying complexity, it can be provided by language workbenches as standard in both cases. For this reason, we do not express an explicit preference for one of these two approaches. Generally, language engineers should check whether decentralization of the import statements makes sense for the specific modeling language.

A8: Import Targets There are several options for realizing the various import targets. Pure artifact referencing, as found in scripting languages, is, however, largely unsuitable. The elements to be integrated via import often appear in nested scopes of an artifact, which must also be referenced in a qualified manner.

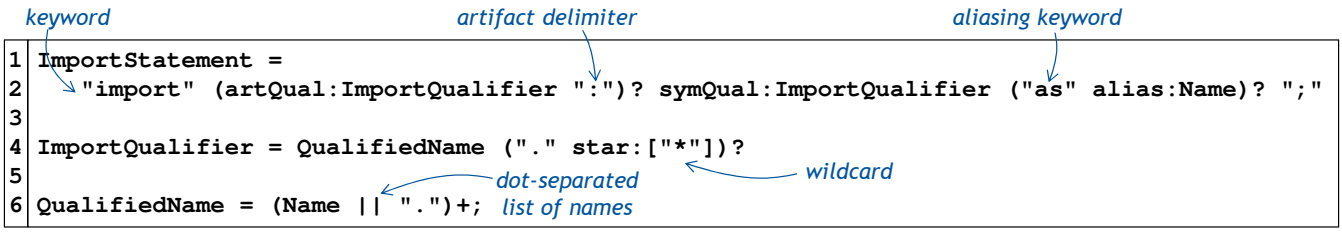

Figure 1: Context-free grammar specifying a standardized version of import statement.

The reverse approach of ignoring artifact names and their location on the file system during imports and referencing exclusively on the basis of the internal scope structure avoids this problem by design. The advantage is that modelers do not have to deal with the model artifact itself but only with its content. Nevertheless, this approach also has a major technical disadvantage: Without artifact references, it is unclear which file the referenced model elements are located in. This leads to a massive frontloading of all potentially relevant artifacts, which represents a significant overhead. Alternatively, each qualified name must be mapped to the respective artifact via preprocessing for efficient adaptive reloading. This approach is generally suitable for larger modeling languages (such as SysML v2, where the strategy is actually applied), where database-like infrastructures are available for model management. For smaller modeling languages and tools, this approach does not seem transferable without considerable overhead, even if it is practical from a usability point of view.

An integrated approach, as common in many programming languages, still allows the artifact to be referenced and seamlessly transition to the qualification of the internal model elements. This represents a compromise between ease of use from the modeler's point of view and efficient reloading of the artifacts. The only disadvantage of this approach is the ambiguity in determining the artifact boundary. The statement `import a.b.c.d.e` provides no information about where the artifact ends, and the internal model structure begins. The required element could be, for instance, in artifact `a.b`, which has a substructure `c.d.e`, or in artifact `a.b.c.d`, which contains element `e` directly. In this case, several artifacts must be reloaded. However, such cases should be arguably rare in reality.

However, this problem does not occur in the last variant, where a systematic distinction is established between the artifact path and internal model qualification. Accordingly, there are two qualified names in the import statement that refer to independent targets. However, rigorously enforcing this dichotomy creates a modeling overhead. It is preferable to offer a combination of implicit and explicit artifact qualification, which allows user-friendly modeling while also enabling a more precise artifact specification if necessary.

A9: Cyclicality Whether cyclic imports make sense depends largely on the objective of the modeling language. Nevertheless, they should not be forbidden in general, as there can certainly be useful applications. At best, a library that offers import statements by default should also support cyclic references, with the option to exclude them by validation rule if they are explicitly not desired. Otherwise, artifact dependency circles should generally be allowed. However, manually excluding the cycles' effects via import guarding is discouraged. This is a technical remnant of more traditional programming languages that modelers should not have to worry about at this level of abstraction. Automatic cycle extraction is, therefore, the technique of choice to support modeling appropriately. Naturally, this is also the most complicated approach algorithmically, but just like in other cases, the functionality can already be provided by default as an incorporable component in a language workbench.

A10: Relative Imports While import statements with absolute qualification are the de facto standard and should be supported in principle, relative qualifications based on the current artifact position are less common. This can result in shorter import declarations if the required element is in a namespace close to the referencing location. There is a minor disadvantage for non-tool-assisted refactorings, as a reference can become invalid in two ways: when the position of the reference source or the target changes. Generally, however, relative imports are an optional feature that can be used reasonably in specific scenarios.

5.2 Assembling a Standardized Import Statement

Based on our evaluation results, we propose an import statement variant with the following syntactic constructs: Wildcards, aliasing, and a combination of an integrated artifact and symbol path as well as an explicit separator. Figure 1 shows a context-free grammar that specifies these import statements. It starts with a corresponding keyword (l. 2) followed by the optional qualifier for the artifact path. Such an `ImportQualifier` (l. 4) consists of a `QualifiedName`, a dot-separated list of simple names (l. 6), and an asterisk to indicate a wildcard. The artifact qualifier requires a delimiter or keyword to mark its range (l. 2). This is followed by the symbol qualifier, which points to the actual internal element via its unique name. The artifact qualifier is declared optional, which means that it does not necessarily have to be used. If the artifact qualifier is omitted, the symbol qualifier is used in the integrative variant instead, containing both the artifact path and the symbol's qualification. Finally, the imported symbol can be renamed using aliasing.

The grammar shown is only intended to serve as an orientation example for the various features and not to prescribe the concrete syntax. Furthermore, not all valid instances are useful. For example, if a symbol qualifier uses a wildcard, there can generally be no alias, as it would be mapped to an undefined set of import names. In a language tool, such context-related conditions would also be

excluded by validation rules. Further constructs, such as visibility, can also be included. However, this was deliberately omitted here, as we do not recommend their use in the standard case. Intrinsic and compound aspects that cannot be mapped directly in the concrete syntax also apply according to our evaluation.

6 Discussion

The presented work analyzes the foundations of import statements in literature and various language realizations. While imports are a well-known concept in programming, there has been no overview of their various realization aspects, as well as their advantages and disadvantages. Our work provides this elaboration concerning modeling languages.

On the one hand, our contribution guides language developers on which aspects are appropriate to consider for their particular import realizations. On the other hand, we also provide a comprehensible set of conceptually reusable, configurable standard import variants. We wish that major language workbenches will offer similar ones in their libraries. Some large language workbenches, such as Xtext [16], MPS [7], and MontiCore [29], already come with standard mechanisms for this. However, providing configurable support for the various standards would be beneficial to relieve language developers of redundant work and offer them a selection of pre-tailored statements for multiple purposes.

Our work is based on our years of experience with software languages in general, as well as their different forms of composition, both on language and model level, but also on modeling as a particular application area. Thus, our results are also subject to threats to validity, as usual when assessing certain features. Threats to internal validity exist in terms of our evaluation of the individual aspects and their characteristics, as these can be influenced by personal experience. To reduce this, we evaluated the aspects dedicatedly against requirements for modeling in general. We also involved colleagues from academia and industry to incorporate their assessment of specific characteristics. Threats to external validity do exist, as we cannot cover all aspects of import statements within the scope of this paper. Also, our classification does not perfectly fit every realization of imports, as there are different implementations that can overlap certain aspects. However, these can be regarded as relatively small, as a variety of state-of-the-art software languages were included in our investigation. Furthermore, our work aims to provide a basis for most modeling languages, for which not every feature, no matter how exotic, is usually required.

Other noteworthy aspects that we have not addressed in detail are, for example, further extension variants in the wildcards. It would also be possible to use several wildcards simultaneously in the qualified path statement, which would open up several resolve branches. Similarly, recursive wildcards could be defined, which include not only all sources at a single level but also all contained substructures. Furthermore, programming languages such as Java [2] differentiate between static and non-static imports. This makes sense in the object-oriented context of this language, but cannot necessarily be mapped to general modeling scenarios, which is why we have deliberately excluded such features. In general, languages can also employ multiple syntactically different import statements, e.g., explicitly separate imports for different sources (e.g., in language aggregation [29]). This practice can be valuable, but it is the same principle with different keywording, which is why we do not consider it in any more detail. While there will always be special cases our work cannot cover, we provide a reasonable baseline for developing import concepts from a common foundation.

7 Conclusion

Language engineering is continuously improving, and modern frameworks already offer a variety of standardized functionalities and design patterns. Our work contributes to this standardization by analyzing the aspects and features of import statements of state-of-the-art software languages. Furthermore, we have explained their implications in the context of MDE, whereby a configurable set of predefined import statements can be derived. Our aim is that this results in a more homogeneous and, therefore, more reusable collection of imports in the future, which can be applied universally across different language workbenches.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

References

[1] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2017. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)* (Taichung). IEEE, 172–179.
[2] Ken Arnold, James Gosling, and David Holmes. 2005. *The Java Programming Language.* Addison Wesley Professional.
[3] Alan Batson. 1965. The Organization of Symbol Tables. *Commun. ACM* 8, 2 (1965), 111–112.
[4] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing Ltd.
[5] Hans Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, Fulvio Tagliabo, Sandra Torchiaro, Sara Tucci, et al. 2013. EAST-ADL: An architecture description language for Automotive Software-Intensive Systems. *Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design* (2013), 456. https://doi.org/10.4018/978-1-4666-3922-5.ch023
[6] Arvid Butting, Judith Michael, and Bernhard Rumpe. 2022. Language Composition via Kind-Typed Symbol Tables. *Journal of Object Technology (JOT)* 21 (October 2022), 4:1–13.
[7] Fabien Campagne. 2014. *The MPS Language Workbench: Volume I.* Vol. 1. Fabien Campagne.
[8] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe (Eds.). 2015. *Globalizing Domain-Specific Languages.* Springer.
[9] KR1442 Chowdhary and KR Chowdhary. 2020. Natural Language Processing. *Fundamentals of artificial intelligence* (2020), 603–649.
[10] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. 2015. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages (LNCS 9400).* Springer, 7–20.
[11] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. 2016. *Engineering Modeling Languages: Turning Domain Knowledge into Tools.* Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series.
[12] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering.* 25–36.
[13] R Kent Dybvig. 2009. *The Scheme Programming Language.* Mit Press.
[14] Robert Ennals and Simon Peyton Jones. 2003. Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-Strict Programs. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming.* 287–298.
[15] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex

Loh, et al. 2013. The State of the Art in Language Workbenches. In *International Conference on Software Language Engineering*. Springer, 197–217.
[16] Moritz Eysholdt and Heiko Behrens. 2010. 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*. 307–309.
[17] J-M Favre. 2005. Languages evolve too! Changing the Software Time Scale. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE, 33–42. https://doi.org/10.1109/IWPSE.2005.22
[18] Peter H Feiler and David P Gluch. 2012. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley.
[19] Matthias Felleisen and Daniel P Friedman. 1986. A closer look at export and import statements. *Computer Languages* 11, 1 (1986), 29–37.
[20] David Flanagan and Yukihiro Matsumoto. 2008. *The Ruby Programming Language*. " O'Reilly Media, Inc.".
[21] William B Frakes and Kyo Kang. 2005. Software Reuse Research: Status and Future. *IEEE transactions on Software Engineering* 31, 7 (2005), 529–536.
[22] I Garcıa, M Polo, and M Piattini. 2004. Metamodels and architecture of an automatic code generator. In *2nd Nordic Workshop on the Unified Modeling Language, NWUML*.
[23] Mark Gardener. 2012. *Beginning R: The Statistical Programming Language*. John Wiley & Sons.
[24] Machtelt Garrels. 2010. *Bash Guide for Beginners*. Fultus Corporation.
[25] Hans Grönniger and Bernhard Rumpe. 2011. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems (LNCS 6662)*. Springer, 17–32.
[26] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. 2012. *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems*. Technical Report AIB-2012-03. RWTH Aachen University.
[27] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal* 37, 10 (October 2004), 64–72.
[28] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. 2008. *The C# Programming Language*. Pearson Education.
[29] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag.
[30] Katrin Hölldobler, Pedram Mir Seyed Nazari, and Bernhard Rumpe. 2015. Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures. In *Domain-Specific Modeling Workshop (DSM'15)*. ACM, 23–30.
[31] Walter L Hürsch and Cristina Videira Lopes. 1995. Separation of Concerns. (1995).
[32] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. 2008. Genesys: service-oriented construction of property conform code generators. *Innovations in Systems and Software Engineering* 4, 4 (2008), 361–384. https://doi.org/10.1007/s11334-008-0071-2
[33] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. 2006. ATL: a QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 719–720.
[34] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. 2022. Correct and Sustainable Development Using Model-based Engineering and Formal Methods. In *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*. IEEE.
[35] Gregor Kiczales, Jim Des Rivieres, and Daniel G Bobrow. 1991. *The Art of the Metaobject Protocol*. MIT press.
[36] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Pearson Education.
[37] Paul Klint, Ralf Lämmel, and Chris Verhoef. 2005. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14, 3 (2005), 331–380.
[38] Stefan Koch. 2011. *JavaScript*. dpunkt.
[39] Vinay Kulkarni and Sreedhar Reddy. 2003. Separation of Concerns in Model-Driven Development. *IEEE software* 20, 5 (2003), 64–69.
[40] Rasmus Lerdorf and Kevin Tatroe. 2002. *Programming PHP*. " O'Reilly Media, Inc.".
[41] MagicDraw Enterprise 2024. *MagicDraw Enterprise*. Retrieved Jul 03, 2024 from https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/
[42] Jeff Meyerson. 2014. The Go Programming Language. *IEEE software* 31, 5 (2014), 104–104.
[43] OMG MOF. 2002. OMG Meta Object Facility (MOF) Specification v1. 4.
[44] Benjamin Muschko. 2014. *Gradle in Action*. Simon and Schuster.
[45] Object Management Group. 2015. XML Metadata Interchange (XMI) Specification, Version 2.5.1. https://www.omg.org/spec/XMI/2.5.1/PDF [Online; accessed 2024-06-05].
[46] Object Management Group. 2017. OMG Unified Modeling Language (OMG UML), Version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF [Online; accessed 2024-06-05].
[47] Object Management Group. 2019. OMG Systems Modeling Language (OMG SysML), Version 1.6. https://www.omg.org/spec/SysML/1.6/PDF [Online; accessed 2024-06-05].
[48] Object Management Group. 2023. Kernel Modeling Language (KerML), Version 1.0 Beta 1. https://www.omg.org/spec/KerML/1.0/Beta1/PDF [Online; accessed 2024-06-05].
[49] Object Management Group. 2023. OMG Systems Modeling Language (OMG SysML), Version 2.0 Beta 1. https://www.omg.org/spec/SysML/2.0/Beta1/Language/PDF [Online; accessed 2024-06-05].
[50] Bernhard Rumpe. 2017. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International.
[51] Ed Seidewitz. 2020. On a Metasemantic Protocol for Modeling Language Extension.. In *MODELSWARD*. 465–472.
[52] Bran Selic. 2003. The Pragmatics of Model-Driven Development. *IEEE software* 20, 5 (2003), 19–25. https://doi.org/10.1109/MS.2003.1231146
[53] Herbert Stachowiak. 1973. *Allgemeine Modelltheorie*. Springer.
[54] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework*. Pearson Education.
[55] Bjarne Stroustrup. 1988. What is Object-Oriented Programming? *IEEE software* 5, 3 (1988), 10–20.
[56] Bjarne Stroustrup. 1999. An Overview of the C++ Programming Language. *Handbook of object technology* (1999), 72.
[57] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. 2019. Import2vec: Learning Embeddings for Software Libraries. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 18–28.
[58] Guido Van Rossum et al. 2007. Python Programming Language.. In *USENIX annual technical conference*, Vol. 41. Santa Clara, CA, 1–36.
[59] Balaji Varanasi. 2019. *Introducing Maven: A Build Tool for Today's Java Developers*. Apress.
[60] Markus Voelter. 2018. Fusing Modeling and Programming into Language-Oriented Programming: Our Experiences with MPS. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I 8*. Springer, 309–339.
[61] Markus Voelter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1449–1450.
[62] Steven Völkel. 2011. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Shaker Verlag.
[63] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. 2013. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.
[64] Dennis Leroy Wigand, Arne Nordmann, Niels Dehio, Michael Mistry, and Sebastian Wrede. 2017. Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case. *Journal of Software Engineering for Robotics* (2017).