university of groningen / faculty of science and engineering

Facilitating Communication about Software Architectures by Extending Architectural Languages

Hugo Kolstee

university of groningen / faculty of science and engineering

University of Groningen

Facilitating Communication about Software Architectures by Extending Architectural Languages

Bachelor's Thesis

To fulfill the requirements for the degree of
Bachelor of Science in Computing Science
at University of Groningen under the supervision of
Dr. Vasilios Andrikopoulos (Computing Science,
Software Engineering, University of Grongingen)
and
Drs. Anja Reuter (Computing Science, University of Groningen)

Hugo Kolstee (s3515249)

March 26, 2023

Contents

Acknowledgments

My research would have been impossible without the aid and support of Drs. Anja Reuter. I am profoundly grateful for the papers she shared with me which steered me in the right direction.

Abstract

Architectural languages are the backbone of a well-structured architecture in software development. Because of the differences in software systems developed, and their differences in architecture, there is a need for different functions within these architectural languages. In the last 20 years, the development of architectural languages has been focused on providing specific functionality rather than generic functionality. It is however unclear how well these languages satisfy the needs of the software engineers and stakeholders. Researching the deficiencies most commonly found in currently used architectural languages resulted in a couple of main findings of which we will focus on one, namely the need for support for communication between the stakeholders, software architects, and software engineers. We tackle this problem by developing an intuitive application that allows users to add, view, and delete architectural description language files. The models contained in these files are mapped to a general representation of architectural languages. This allows users to view architectural description models of different architectural languages in the same representation. On top of that, we allow users the functionality to add a description to these files and the ability to add annotations to parts of these models.

Abbreviations

AADL	Architecture Analysis & Design Language
AD	Architectural Description
ADs	Architectural Descriptions
AL	Architectural Language
ALs	Architectural Languages
AOM	Aspect-Oriented Modeling
CCPS	Cloud-enabled Cyber Physical System
CCPSs	Cloud-enabled Cyber Physical Systems
CPS	Cyber Physical System
CPSs	Cyber Physical Systems
OMG	Object Management Group
SQL	Structured Query Language
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Introduction

In software engineering, an architectural language is a language or model to describe and represent the architecture of a system. The architecture of software represents the structures of the software. Software architects are the developers of the architecture of software. Often software developers or teams of software developers are responsible for structures within the architecture.

While current architectural languages' main functionality is modeling the architecture, they often neglect the communication side. Communication between stakeholders and the software architects is seen as insufficiently supported, according to Malavolta, et al [1]. This paper also highlights the tradeoff between analytical functionality and support for communication. Architects that use more formal languages report more lack of communication from the AL they use than architects that use informal languages. Moreover, this is also the case for analytical functionality, but the other way around.

The deficiency of support for communication in architectural languages is the issue we propose to tackle. In what way can we provide more support for communication in existing architectural languages? And can this be done without reducing the analytical functionality of these languages?

To address these questions we propose a web application that allows the software architects and stakeholders to add, view, or delete architectural language files of different architectural languages. The application then shows a general representation of the models within these files.

The target audience will be stakeholders for which it is important to communicate architectural models. This can be nontechnical stakeholders or technical stakeholders like software architects themselves. For example, information for non-architects that is important to be communicated might be how components in the model work together or contact information of the people responsible for parts of the system, and for architects, it could be patterns the components form or other technical data associated with the components.

With this tool, the users would be able to add notes, descriptions, and contact information of people responsible for parts of the architecture, to architectural language files. The application will map the architectural models of architectural languages to a general data model and then visualize it in a general, simple, and easy-to-understand way. The general representation's sole purpose is to support the communication of these models. This way the analytical functionality of the original files will remain unchanged.

To summarize, this thesis focuses on the following problem:

Q1. How can architectural descriptions be extended to accommodate information that supports stakeholder communication?

What follows in the next chapter is detailed research into the deficiencies of current architectural languages. Doing this research we acquired knowledge of the state of the art of architectural languages and their shortcomings. The goal of the research was to have enough knowledge to find a feasible project to contribute to one of the shortcomings of architectural languages. In the chapter after that, the tools and technologies that are used to develop the web application are specified and explained. Following in chapter 4 is the methods chapter. The methods chapter is described how we put the tools and technologies of the previous chapter to use to create the application. In chapter 5 the application is shown, along with the new model's appearance. An evaluation of the application is given in the second-last chapter, and the thesis is concluded in the last chapter.

2 Background Literature

2.1 Architectural Descriptions, Views and Viewpoints

Software architectural descriptions (ADs) are a set of practices, techniques, and types of representations or visualizations that software architects use to express the architecture of software. This can be many things, from informal drawings to text to formal architectural languages. These descriptions are often organized into specific views, where each view represents a set of system concerns. A view has its viewpoint, which specifies the notations and modeling techniques that should be used to express the architecture to the responsible stakeholders. Some examples of a viewpoint include functional, logical, requirements, implementation, performance, security, and information/data. Architectural Languages (ALs) are languages used to represent ADs.

2.1.1 Viewpoint-Oriented Software Development

Tackling the problem of increasingly complex software in the 1990s, several researchers explored the first steps of viewpoint-based software architecture, then phrased it as "the multiple perspectives problem" [2] [3]. This term referred to the problem that, when developing large systems with multiple actors responsible for parts of the system, the organization of development might benefit from a common framework that helps every actor represent its perspective. Among other issues, when perspectives of actors intersect or overlap but each actor uses their representation schemes and development strategies for their part of the system, this framework helps organize the differences to support development cooperation. In this framework, viewpoints are seen as objects with their own identity, state, and behavior. The viewpoint object contains partial knowledge of how to develop that particular part of the system and is seen as an "actor", "role", or "agent" that has its own "perspective" or "view". This early idea on viewpoints has developed its way over the years into the current viewpoint-oriented software engineering.

In 2000, the IEEE-SA Standards Board approved the IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems [4]. This standard introduced a recommendation of practice involving the concept of using different views to describe different perspectives of the architecture of software. It also introduced the concept of a viewpoint to designate the means used to construct individual views. These views can then be used to describe different perspectives, of the architecture of software, also known as architectural descriptions. In the standard a view and viewpoint are defined as follows:

- A view is a representation of a whole system from the perspective of a related set of concerns.
- A view has its viewpoint, which is a specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

In other words, a viewpoint represents a collection of patterns, templates, and conventions for constructing one type of view. Additionally, a viewpoint defines the stakeholders whose concerns are reflected in the viewpoint. The viewpoint's role is to define how to focus on particular aspects of the architecture. The view then represents a part of the architecture through that particular viewpoint.

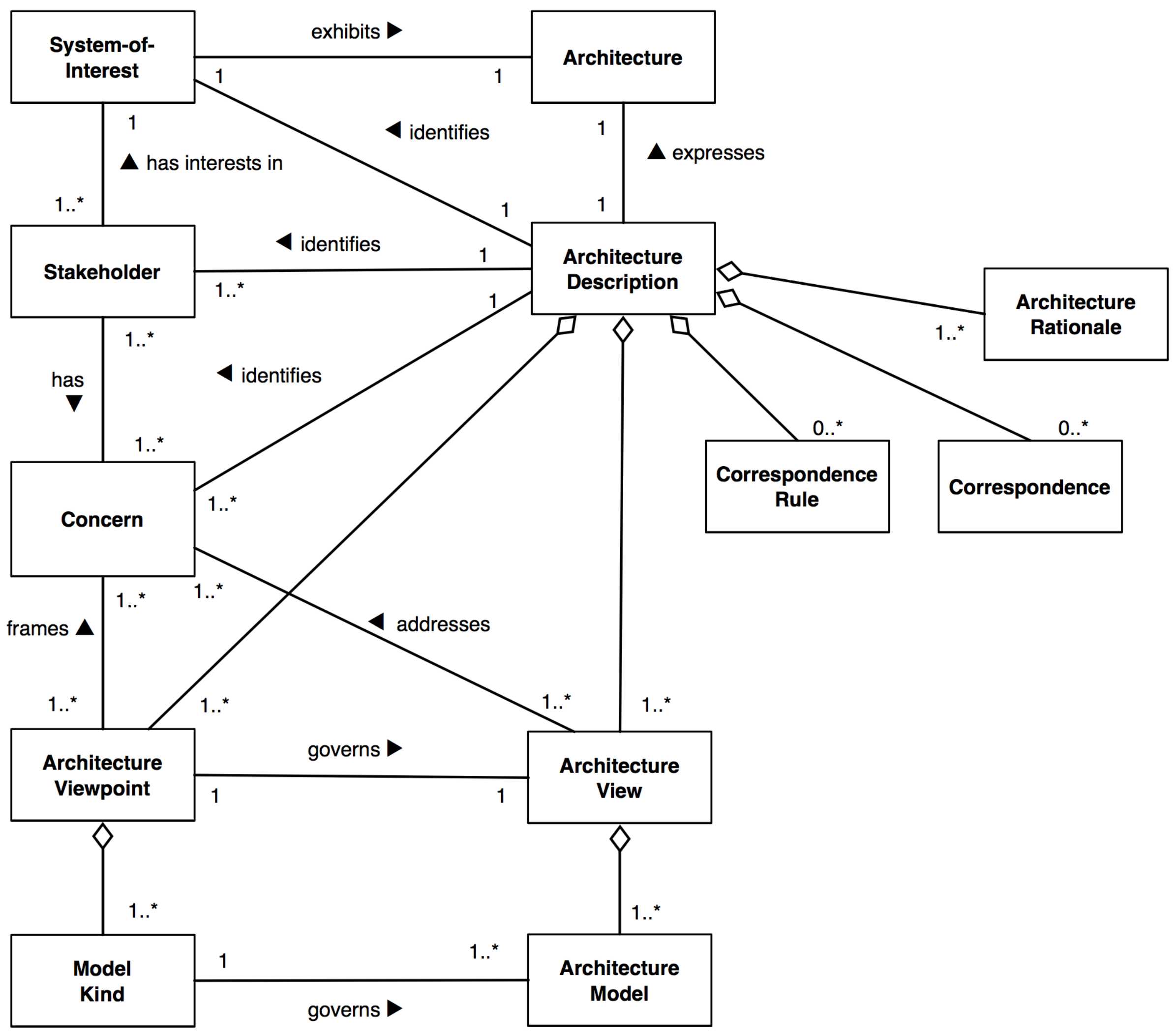


Figure 1: Conceptual Model of Architectural Description (from [5])

Published in 2000, the IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems [4] does not specify specific viewpoints. Rather, it introduces the concept of a viewpoint to designate the means to construct individual views. Even though the 2011 IEEE International Standard for Architecture descriptions [5] also defines a guide to create viewpoints, it also describes five viewpoints for specifying Open Distributed Processing Systems. Distributed Processing Systems are complicated systems where components are located on different networked computers, and therefore, often a large system with subsystems. The five viewpoints [5] defined are the Enterprise viewpoint, Information viewpoint, Computational viewpoint, Engineering viewpoint, and Technology viewpoint. Each viewpoint has its own set of concerns. Together, these 5 viewpoints aim to cover crucial concerns adequately, but the standard does indicate that an AD does not need to be limited to the five predefined viewpoints. The AD may include additional viewpoints and views, as needed. Each viewpoint frames these concerns, in summary:

- Enterprise viewpoint: the purpose, scope and policies of the system and the roles played by the system.
- Information viewpoint: the semantics of information and information processing in the system.

- Computational viewpoint: a functional decomposition of the system into objects which interact at interfaces.
- Engineering viewpoint: the mechanisms and functions required to support distributed interaction between objects in the system.
- Technology viewpoint: the selection of implementable standards for the system, and their implementation and testing.

When constructing an AD, an architect selects the viewpoints needed. The architect then constructs the views based on these viewpoints. Currently, there are predefined viewpoints, defined by standards, ALs, libraries, catalogs, or other sources. The goal of the 2000 IEEE standard was to make viewpoints reusable and integrable, similar to a template. After the viewpoint has been chosen for a particular AD, it has to be slightly changed to fit the specific stakeholders and concerns needed to represent the system. The specific stakeholders and concerns can not be fully specified for each existing system. The 2011 IEEE standard guides the creation of viewpoints more to a predefined direction with more requirements.

The 2011 standard [5] also defines a Conceptual Model of Architectural Description (Figure 1) according to the recommended practice of the defined viewpoint-and-view-oriented architectural development of software. Figure 1 shows the key concepts and their relationships. In summary, an architectural description is formed by views. A view is governed by a viewpoint, which is based on a particular set of concerns of stakeholders that have an interest in the system. Multiple models of parts of the system can form one view.

2.2 Architectural Languages

Currently, it is common practice to use architectural languages (ALs) to represent an AD. These architectural languages can be put into one of three categories: formal architecture description language, UML-based languages, and box-and-line informal drawings.

Before viewpoint-oriented architectures, the development of ALs was focused on making ideal general-purpose AL. However, from 2000 and onwards [1], the focus has been on creating support for different specific requirements because of the increasing complexity of the systems developed.

Nowadays software architectures focus on a lot of different functions. These functions are then represented using views. Some of these functions include (but are not limited to):

- Documenting and communicating design decisions and architectural solutions.
- Analysis techniques like testing, model and consistency checking, and performance analysis.
- Code generation purposes in model-driven engineering
- Risks and costs estimation.

In a study by Woods [6] the requirements of an architectural language from the perspective of an architect are identified. This study was done with software architects in the field of Information Systems. The requirements were identified as follows:

- Better support for multiple views (functional, informational, concurrency, and deployment).
- Direct support for describing domain-specific concerns.
- Satisfactory tool support.
- Support for incremental adoption.
- Support for reusing architectural models.

Currently, according to Malavolta et al [1], the formal languages most used in the industry are AADL, ArchiMate, and industry ad hoc notations. These three languages take up 41 percent of all architectural languages used according to the survey [1]. This survey, done with 48 professional software architects, selected to represent different architecture languages and functionality, also identified two needs from those architects relating to architectural languages. These needs are analysis and communication. Currently, formal architectural languages are mainly analytically focussed while informal architectural languages are more suitable for communication. Respondents to the survey [1] that indicated a need for analytic support also indicated the limitations of the communication aspects with stakeholders. This indicates that currently there is a tradeoff between support for communicating the language with non-architects and analytical support when dealing with technical and developmental support [1].

Communication with non-architect stakeholders proves to be an important part of architectural languages. One of the reasons for the growing amount of architectural languages is stakeholders' concerns [1]. The language has to capture the changing requirements set by stakeholders in the evolving and adapting environment of software development. This is one reason why a perfect general-purpose architectural language is not likely to exist [7][8][9][10].

Woods and Hilliard [11] associate 5 deficiencies that obstruct the widespread adoption of architectural languages:

- ALs often make restrictive assumptions that might be inappropriate.
- ALs are developed with a single-view orientation instead of a multiview orientation.
- ALs are not supported sufficiently by required tools.
- ALs often are general purpose with few domain-specific entities which are important to the architects.
- ALs are not adopted in industrial settings.

We now focus the research on the lack of support for communication in architectural languages. To emphasize the shortcomings of ALs support for communication, we highlight one of the main findings of [1] (finding F2). The finding is as follows: "ALs should combine features supporting both communication and disciplined development. We call this *introvert versus extrovert nature of architects role*". The implications of this finding according to the article are: "Major academic research efforts have been dedicated so far to formal and domain-specific ALs, while industry turns to informal ALs for communication purposes (considered as a primary need in practice). This finding emphasizes the need to further improve ALs in this direction: ALs need to be simple and intuitive enough to communicate the right message to the stakeholders involved in the architecting phase, but they shall also enable formality so as to drive analysis and other automatic tasks."

This finding has motivated the development of the application. By offering the functionality to visualize different ALs in a general simplified intuitive representation we hope to add to the communicative abilities of different ALs. The analytical ability will not be impaired because this is just a mapped conversion of the original AL models. The original AL files will be untouched.

2.3 Architectural Languages in Focus

Following is a description of ALs we focus on as examples for our project, namely UML, AADL, and Archimate. These ALs are selected because according to a survey [1] 86 percent of industry experts with wide ranges of experience and organizations use UML or a UML profile. Because UML is not a formal AL we also look at the two most used formal ALs. The most used formal ALs were found to be Architecture Analysis & Design Language (around 16 percent of respondents) and ArchiMate (around 11 percent). Furthermore, around 12 percent of respondents use formal ALs exclusively, around 35 percent use a mix of formal ALs and UML, and around 41 percent use UML exclusively.

2.3.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a universal modeling language used for modeling systems in the field of software engineering. It aims to provide a standardized way to visualize the design of a software system. UML was originally developed because of the desire to standardize different design notations and software design methods. The language features 13 types of diagrams, each with their own purpose. These diagrams are separated into 3 categories, namely Structure, Behavior and Interaction Diagrams.

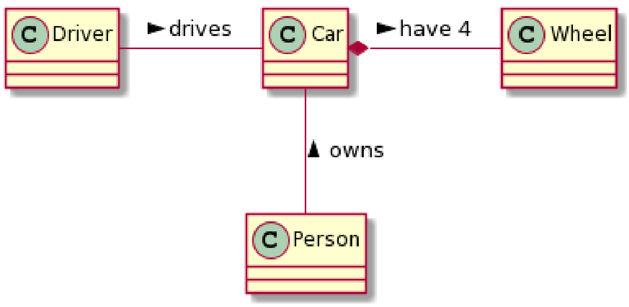


Figure 2: A basic UML class diagram made using the PlantUML UML editor[12].

The Object Management Group (OMG) adopted UML as a standard in 1997. In 2005, the International Organization for Standardization (ISO) also published UML as an approved ISO standard [13]. UML has been an ISO standard ever since, only regularly revising the standard to cover the latest revisions of UML.

UML uses different notations for different relations between components. An example UML class diagram model, which models the structure of classes in a software system, is shown in Figure 2. In the figure, the class wheel is shown to be a composite of the class car. The relation between the driver class and the car class, as well as the person class and the car class, is an association. This means that the classes need to communicate with each other, but there is no special directional relationship.

2.3.2 UML 2.0 viewpoints

UML lets users define model systems by offering support for separating concerns through 4 separate viewpoints [14]:

- Models produced from the static structural viewpoint describe the system's structural aspects. Class models are examples of descriptions produced using this viewpoint.
- Developers use the interaction viewpoint to produce sequence and communication models that describe the interactions among a set of collaborating instances.

- The activity viewpoint is used to create models that describe the flow of activities within a system.
- The state viewpoint is used to create state machines that describe behavior in terms of transitions among states.

UML 2.0 does not support modeling systems from user-defined viewpoints. However, there is work being done on UML-based aspect-oriented modeling (AOM). AOM aims to provide tools to describe systems from user-defined viewpoints [15].

2.3.3 AADL

Architecture Analysis & Design Language (AADL) is a formal architectural language that is standardized by the Society of Automotive Engineers (SAE) [17]. It was designed to support a full Model-Based Development lifecycle with the support for multiple forms of analysis.

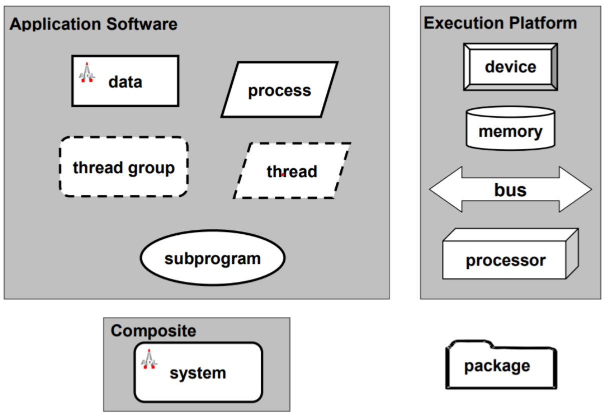


Figure 3: AADL graphical notation[16].

Originally developed in the field of avionics the language was formerly known as the Avionics Architecture Description Language. It was developed in this field because of the complex embedded systems in avionics, like CPSs, which are often real-time systems. A real-time system is a computer system that controls, responds to, and monitors the external environment in real-time. The system uses sensors, actuators, or other input/output methods to measure and respond to the environment. Because of the importance of hardware components in embedded real-time systems, AADL allows users to model both software and hardware components.

The language can be extended by properties defined by the user itself. These user-defined properties can help users specify properties they deem required in their model. Another way AADL is extendable is through the use of language annexes. These annexes allow for user-defined behavior using a set of pre-defined tools. The currently defined annexes are Error-Model, Behavior, ARINC653, and Data-Model Annex.

- Error-Model Annex [18]: Automation of safety analysis methods by supporting them through analyzable architecture fault models. It allows the user to annotate system and software architectures expressed in AADL to be annotated with hazard, fault propagation, failure modes, and effects due to failures, as well as compositional fault behavior specifications to facilitate incremental and scalable automated safety analysis.
- Behavioral Annex [19]: Adds further semantic and syntactic definitions to express components' behavior. It mainly consists in providing an enhanced data type system and a way to express executable parts of threads and subprograms in a portable way that is without using an implementation language.

- ARINC653 Annex: Provides guidance on a standard way of representing avionics systems.
- Data-Model Annex: Provides descriptions of modeling user-specified data constraints.

2.3.4 ArchiMate

ArchiMate is an open and independent AL that is mainly used in Enterprise Architecture Modeling. The language provides a set of default iconography and a set of entities and relations for describing, analyzing, and communicating concerns in the field of Enterprise Architecture. Enterprise Architecture refers to the structures and behaviors of a business, and how they interact with data.

ArchiMate is an Open Group standard. The Open Group is a global standardization organization that has over 800 members including NASA, Philips, IBM, and Huawei. The organization focuses on developing technological standards and certifications.

The viewpoints of ArchiMate are grouped into four categories:

- Basic Viewpoints: Used for concepts of the three main layers of Business, Application, and Technology.
- Motivation Viewpoints: Allows for modeling the motivational perspective of the architecture.
- Strategy Viewpoints: Used for modeling the strategic perspectives of the business to describe strategic directions.
- Implementation and Migration Viewpoints: Used for modeling management concerns when changing the architecture. One example is transitioning from baseline to target architecture.

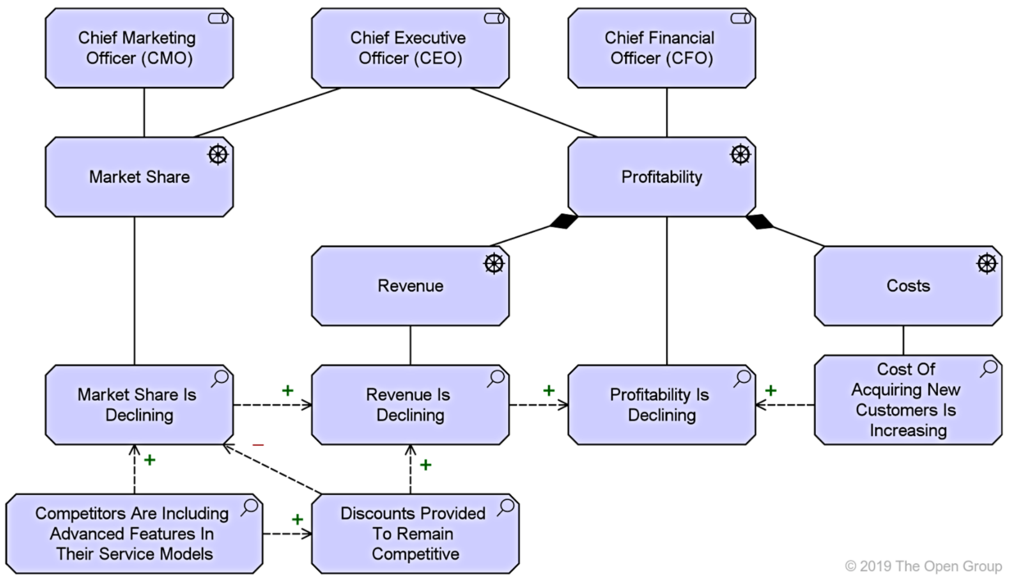

Chief Marketing Officer (CMO)
Chief Executive Officer (CEO)
Chief Financial Officer (CFO)
Market Share
Profitability
Revenue
Costs
Market Share Is Declining
Revenue Is Declining
Profitability Is Declining
Cost Of Acquiring New Customers Is Increasing
Competitors Are Including Advanced Features In Their Service Models
Discounts Provided To Remain Competitive
+
−
© 2019 The Open Group

Figure 4: An example ArchiMate assessment model[20].

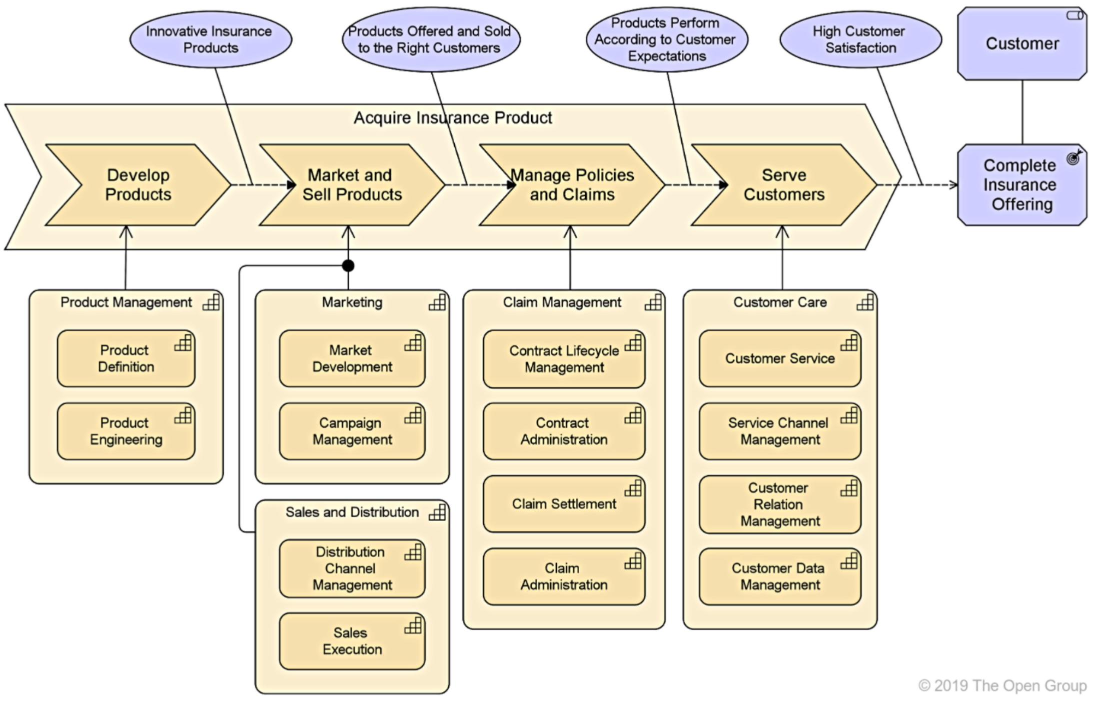


Figure 5: An example ArchiMate strategy model[21].

Figure 4 shows an ArchiMate assessment model. The Chief Marketing Officer (CMO) is concerned with the market share, the Chief Executive Officer (CEO) is concerned with the profitability and market share, and the Chief Financial Officer (CFO) is concerned only with the profitability. The profitability is composed of two other concerns, namely Costs and Revenue. The bottom two rows are assessments. These assessments are associated with concerns and may positively or negatively influence each other. For example in the model, the market share decline results in revenue decline, which in turn results in the profitability declining.

Figure 5 shows an Archimate Strategy model. The figure models a high-level value stream. The different stages is served by a number of capabilities. Between these stages, we see the value flows with associated value items. At the end of the stream is the outcome which is what the stream realizes for a particular stakeholder.

3 Methods

3.1 Data model

Designing the general data model was done by keeping in mind that ALs can be reduced to a set of components with a set of relations between these components. What differentiates ALs is how these components and relations are defined. For example, in a UML Class Diagram, the classes are the components, and there is a set amount of relations. This is a very basic example with only one component type, but the fundamentals are the same for other ALs.

To not make the general representation too general, we want to keep the information associated with the individual Components. This is separated into two groups, namely Attributes and Operations. Attributes can be seen as what information is of importance to describe the Component. Operations can be seen as what actions the Component is capable of. For some ALs, it is important to identify what type of information/data goes into or returns from an Operation. This is why we added the Parameter in the structure of our datamodel. The Parameter is owned by an Operation.

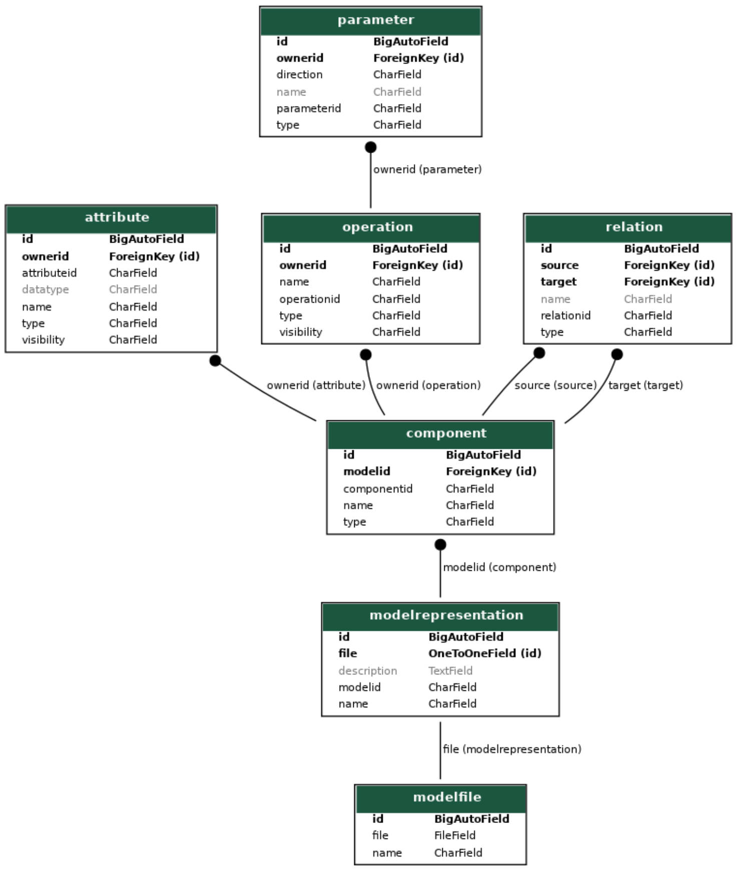


Figure 6: The generalized Datamodel

3.2 Mapping

The mapping of architectural language models to a general representation is done according to Table 1. The current application supports UML Class Diagram mapping, and when extending the application to support AADL or Archimate it should be implemented as shown in the table.

Figure 7 shows a summary of AADL elements. The graphical notation of these components is shown in Figure 3. In AADL, components are defined by type and implementation. The Component Type contains the information of the component's externally visible characteristics and represents the software and hardware components of a system. The Component Implementation specifies internal information about the component. The internal information includes subcomponents, calls and connections with the features of those subcomponents, modes that represent operational states, and properties. Properties can be predefined, like the runtime of a thread, or user-defined for additional support for analysis.

Elements in the table that we want to elaborate on are the Parameter and Parameter Connection elements under the AADL mapping. A Parameter in AADL is used to model software components that use a parameter. For example a C Function. The Component is then the C Function that has a parameter that is indicated in the general model as an Attribute. The second element is a Parameter Connection. A Parameter Connection in AADL is used when a user wants to model threads that run certain functions that require parameters. It is then required to model a parameter connection for the Thread or Thread Group. Both these elements are highlighted in Figure 7.
Figure 8 shows the general metamodel for the behavior and structure elements of ArchiMate. Struc-

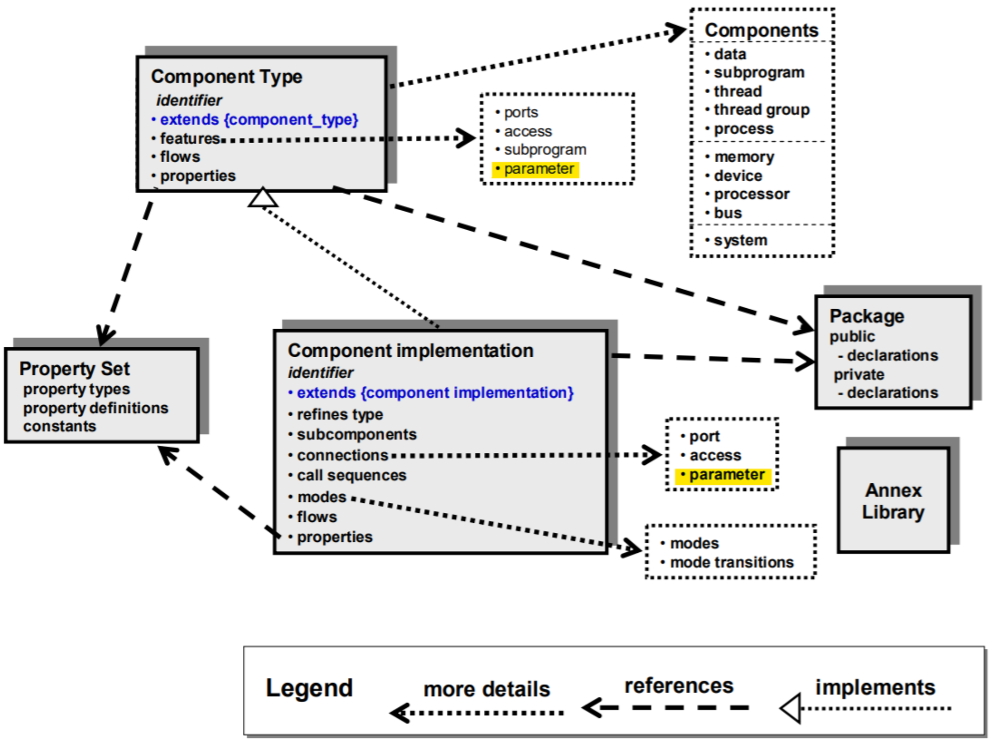


Figure 7: A summary of AADL elements [16]. Highlighted in yellow are the Parameter and Parameter connection elements.

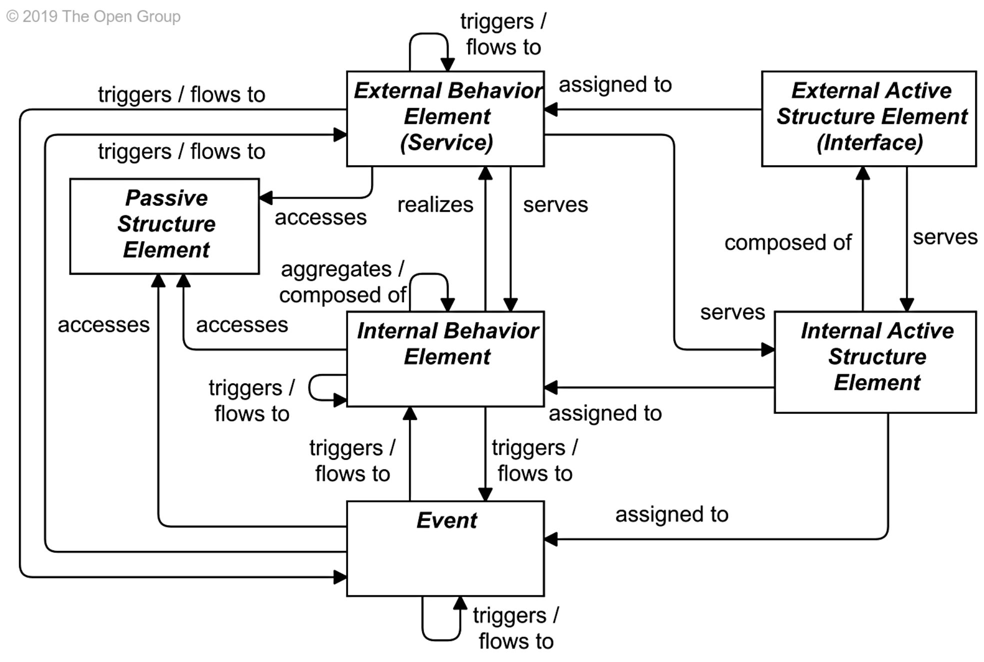


Figure 8: ArchiMate Behavior and Structure Elements Metamodel [22].

ture elements are divided into active and passive structure elements. Active structure elements are the elements that can perform a behavior. These structural elements are subdivided further into internal and external active structure elements. Internal active structure elements are elements that realize the behavior; for example business actors or application components. The elements that expose this behavior to the environment are the external active structure elements. These elements are also called interfaces. An interface provides an external view of the service provider and hides its internal structure. Lastly, the passive structure element is an element on which behavior is performed. Examples of this type of element are information or data objects.

Behavior elements can also be subdivided into internal and external elements. The behavior type represents the dynamic aspects of the enterprise. Internal behavior elements represent a unit of activity that an active structure element can perform. An external behavior element, which is called a service, represents an explicitly defined externally exposed behavior. Then there is one more behavior element type, namely an event. An event represents a change in the state of the system.

Elements types that are not shown in this fragment of the complete metamodel are the motivation element and location element. The motivation element represents the context or reasoning behind the architecture of an enterprise and the location element is simply a location where concepts are located or performed.

For two examples of ArchiMate models see Figure 4 and Figure 5.

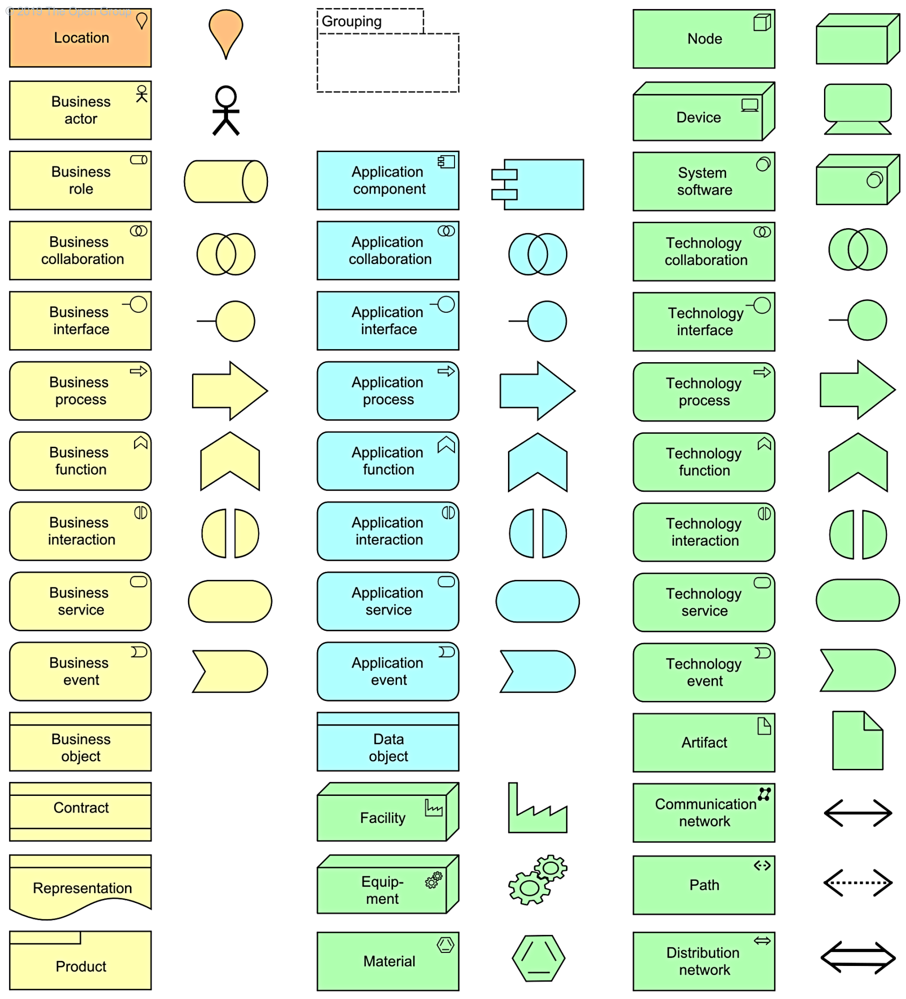


Figure 9: A summary of AADL elements [16]. Highlighted in yellow are the Parameter and Parameter connection elements.

Table 1: Mappings for Example Architectural Languages

General Data Model	UML Class Diagram	AADL[23][24][16]	ArchiMate[22][25]
Component	Class	*Software components:*	*Generally:*
		Process	Internal Behavioral Elements
		Thread	Passive Structure Elements
		Thread Group	Internal Active Structure Elements
		Subprogram	
		Data	*By layer:*
			Business Elements
		Hardware components:	Motivation Elements *(See Figure 4)*
		Device	Application Elements
		Processor	Technology Elements
		Memory	Strategy Elements *(See Figure 5)*
		Bus	Composite Elements
			Implementation Elements
		Composite components:	Migration Elements
		System	Physical Elements
		See Figure 3 for graphical notation	*Other:*
			Location
Relation	Association	Component Implementation to Component Type: Generalization	*Structural Relationships:*
	Aggregation	Subcomponents relation	Realization
	Composition	Port connections	Assignment
	Generalization	Component access	Aggregation
	Realization	Property association	Composition
		Parameter connection	
			Dependency Relationships:
			Association
			Influence
			Access
			Serving
			Dynamic Relationships:
			Triggering
			Flow
			Other Relationships:
			Specialization
			AND/OR Junctions of Relations treated separately
Attribute	Class Attribute	Component properties	
		Data port	
		Event port	
		Bus access	
		Data access	
		Parameter	
Operation	Class Method	Service subprogram	
Parameter	Method Parameter (in/out)		

4 Tools and Technologies

4.1 The Django Web Framework

A prototype application allowing for adding communication aspects to existing architectural models was developed as a web tool using the Django web framework. Django is a high-level Python web framework used to build web applications and services. The framework is based on a Model-Template-Views architectural pattern and provides support for the creation of database-driven websites. Additionally, one of its main goals is to allow its users reuse-ability of components. This can be seen in the recommended architecture, where applications are sections of a project. These applications are standalone and, if done according to the recommendation, can be copied and used in any other project.

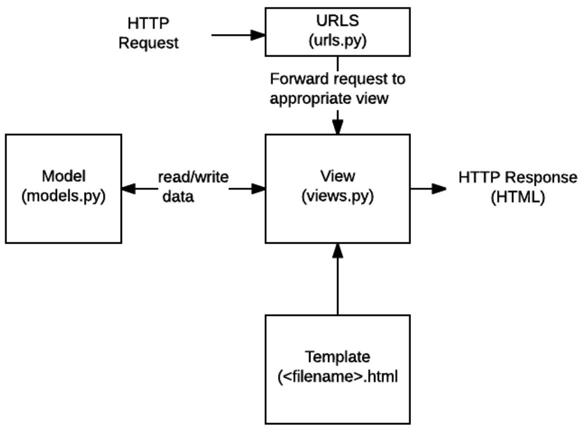


Figure 10: A visualization of the django Model-Template-View architecture [26].

In the Model section of the application, the data model used in the database is specified. Moreover, the tool developed also calls functions to map given XML files into the database from this section. Manipulation and definition of the database and its components is done in the Model section of Django.

The Views section in the application defines what data is provided for the template for the current web page. The template manages what is displayed on the page. This data often consists of data fetched from the database. The application also generates the graphical visualization of the selected model in this section.

Templates in Django are HTML files that specify how the data provided by the view section is shown on different pages. It is not possible to call functions or do calculations in the template files, so any required calculation or function data also has to be provided by the Views section.

4.2 Mapping Architectural Language Generated XML Files

To map an AL model file to the general representation the usage of parsing and processing Extensible Markup Language (XML) is used. XML is a markup language that is extendable by the user to define encoding for documents by following a set of rules. A markup language is a language that uses tags to define elements in the document which makes it easy to visually distinguish the content. XML is mostly used for processing document information in a formatted way. The language is a standard of the World Wide Web Consortium (W3C) the Standard Generalized Markup Language[27].

The three ALs described earlier in this thesis, namely UML, AADL, and Archimate, all allow for exporting of models in XML format. For UML, this is the case using two popular editors that we will be using to generate XML files.

First of all, PlantUML. PlantUML is a text-based UML diagram creation tool. It supports all UML diagrams and is widely used. It allows for export in an XML Metadata Interchange Format (XMI), which is based on XML and can be processed identically.

The second editor is StarUML. StarUML is a UML tool that uses a Graphical User Interface to create UML diagrams. StarUML is one of the most widely used graphical UML editors and also supports all UML diagrams. Using an extension we can export and import the diagrams in an XMI format. The XMI export command can be called with a special flag that creates an XMI file that can be imported to StarUML 2.0. For the application, we also added functions to map StarUML 3.0 generated XMI files because they are differently formatted. Similar import functionality could be implemented for other.

4.2.1 The ElementTree XML API

The ElementTree XML API is a Python3 standard library module. Because of the hierarchical nature of XML files (shown in Listing 1), the files can easily be represented in a tree structure. The Elementree module does just that. It lets users parse XML files into a tree that is easily explored and processed. To traverse the tree the module offers functions to call on the root of the original tree, or a root of any of the subtrees that make up the complete tree. Functions like *find* or *findall* return a filtered node object or list of node objects in the structure that conform to the filter. This allows users to then use these objects as new roots for subtrees, or to call functions like *get* on these objects to extract specific data.

```
...
    <UML:Class name="Mission" namespace="model1" xmi.id="cl0025">
        <UML:Classifier.feature>
            <UML:Attribute name="name : string" xmi.id="att120"/>
            <UML:Attribute name="type : char" xmi.id="att121"/>
            <UML:Attribute name="description : string" xmi.id="att122"/>
            <UML:Operation name="actionEventCheck()" xmi.id="att123"/>
        </UML:Classifier.feature>
    </UML:Class>
    <UML:Association namespace="model1" xmi.id="ass124">
        <UML:Association.connection>
            <UML:AssociationEnd aggregation="aggregate" association="ass124" type="cl0004"
    xmi.id="end125">
                <UML:AssociationEnd.participant/>
            </UML:AssociationEnd>
            <UML:AssociationEnd association="ass124" type="cl0005" xmi.id="end126">
                <UML:AssociationEnd.participant/>
            </UML:AssociationEnd>
        </UML:Association.connection>
    </UML:Association>
...
```

Listing 1: An sample of XML code, highlighting the hierarchical structure of the language. The sample code shows the formatting of a class and a relation of a PlantUML generated XMI file. Taken from one of the sample files used for the application.

In the application, the ElementTree module is used to first find all models specified in the XML file. Then on each of the found models, a call is made to look for components. Each component often has specific attributes and/or operations and relations. However, this depends on how the data is formatted in the XML file, and can't be assumed to be generally adopted by ALs that allow XML exporting. Extending the application to support more ALs can be implemented by extending the application with format-specific functions to parse the XML files in a similar way, and then map the data to fit the general data model.

4.3 MySQL Database

For the application, a MySQL database is used. The MySQL software provides the application with a fast, multithreaded, multi-use Structured Query Language (SQL) database server. Our model data is saved to the database according to the data model for the general representation we use to represent AL models. The data model is given in Section 3.1. Additionally, in the database, login information is stored.

4.4 Graphviz

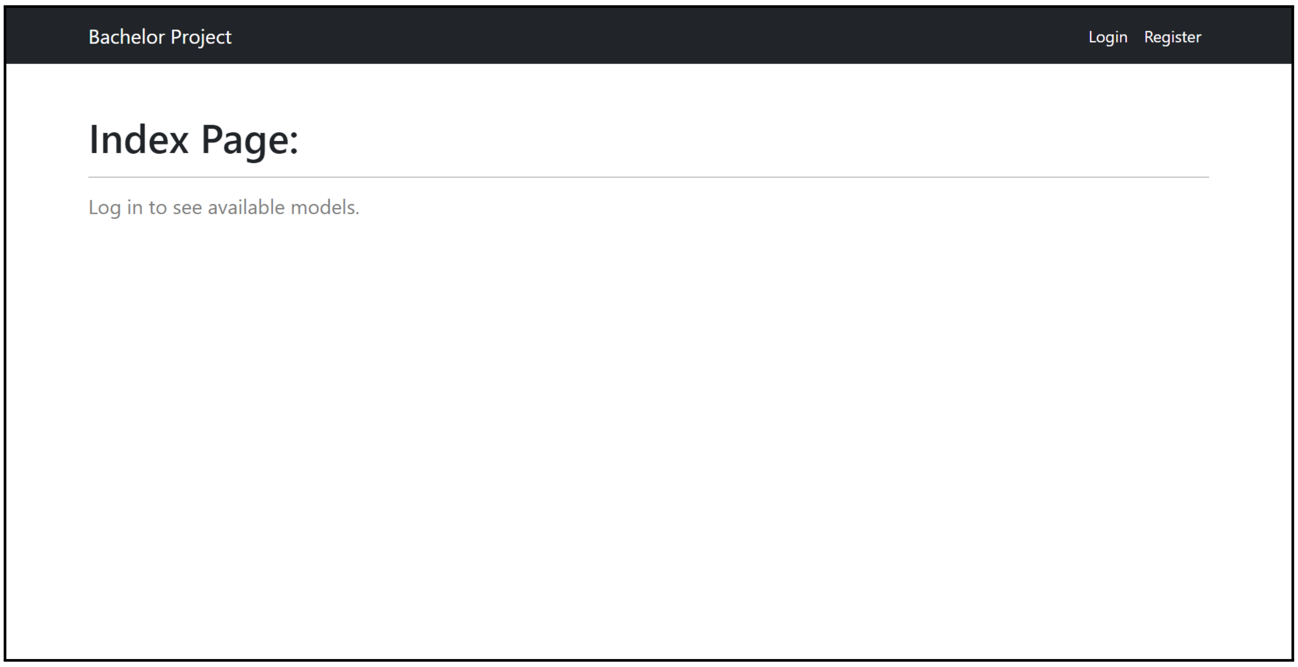
Graphviz for Python is a package that allows the application to create graph representations in the DOT language. It lets users build graphs from user-specified nodes and edges and subsequently manipulate both the graph its content and its visual representation.

5 User Interface

The application itself can be summarized as follows. The home page of the web application is a list of models currently available models. This is shown in Section 5.1 and is called the Index Page. The name of the model is a link that takes the user to the Detail Page of the model shown in Section 5.2. On this page, the generalized model representation is shown. Additionally, the user can add or update the text-based description of the model. Furthermore, below the description is a list of all elements in the model. Here all the components are also listed, along with button a button to add an annotation, or buttons to update or delete the annotation if it already exists.

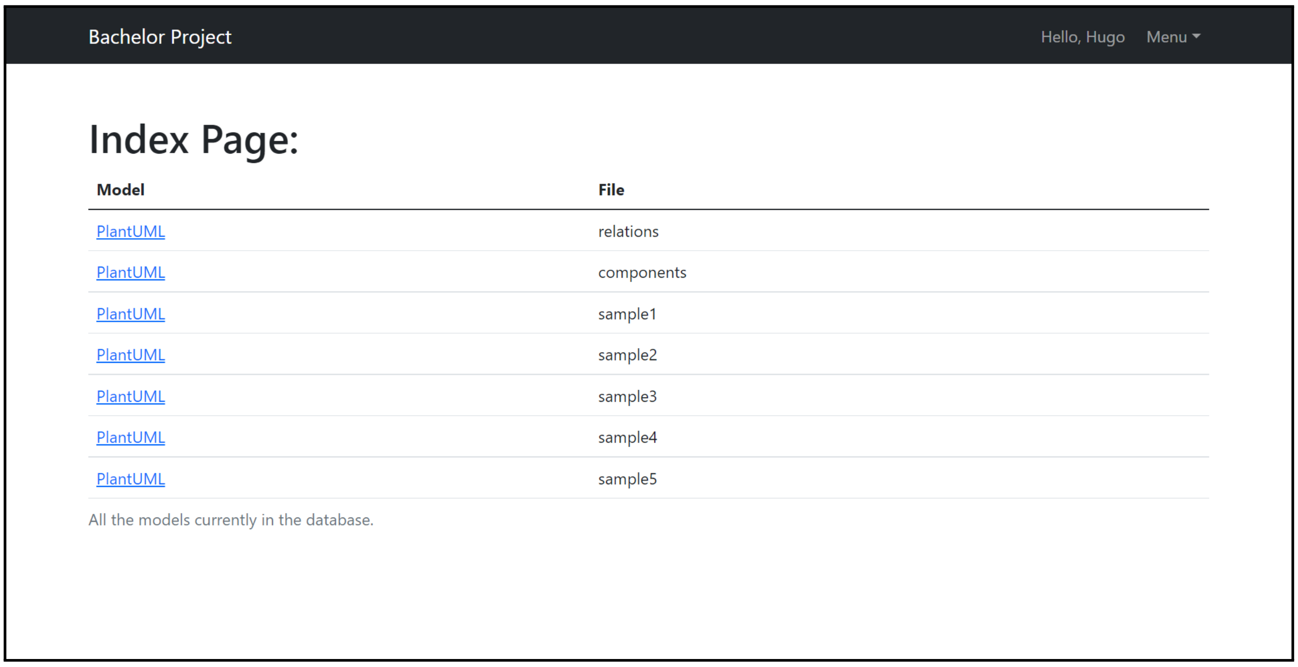
The way the files are managed is in the drop-down menu located at the top right of every page. In this menu are links to two file-related pages shown in Section 5.3. The first is the Upload File link. This page allows the user to specify the name associated with the file, and upload a file to be processed by the application. Once the upload is successful, the user is redirected to the All Files Page. This is also the second entry in the drop-down menu. This page shows a table that lists all the files currently in the database. The first column in the table is the name of the file, and next to that is the Download column. This is where the filename is shown, and when clicked, the user's system downloads the file to the computer. The last column is where the Delete buttons for each file are located. Once clicked, the file is deleted from the database along with the locally saved file. By default, Django saves files locally in the application where the server is running.

5.1 Index Page



Index Page:

Log in to see available models.



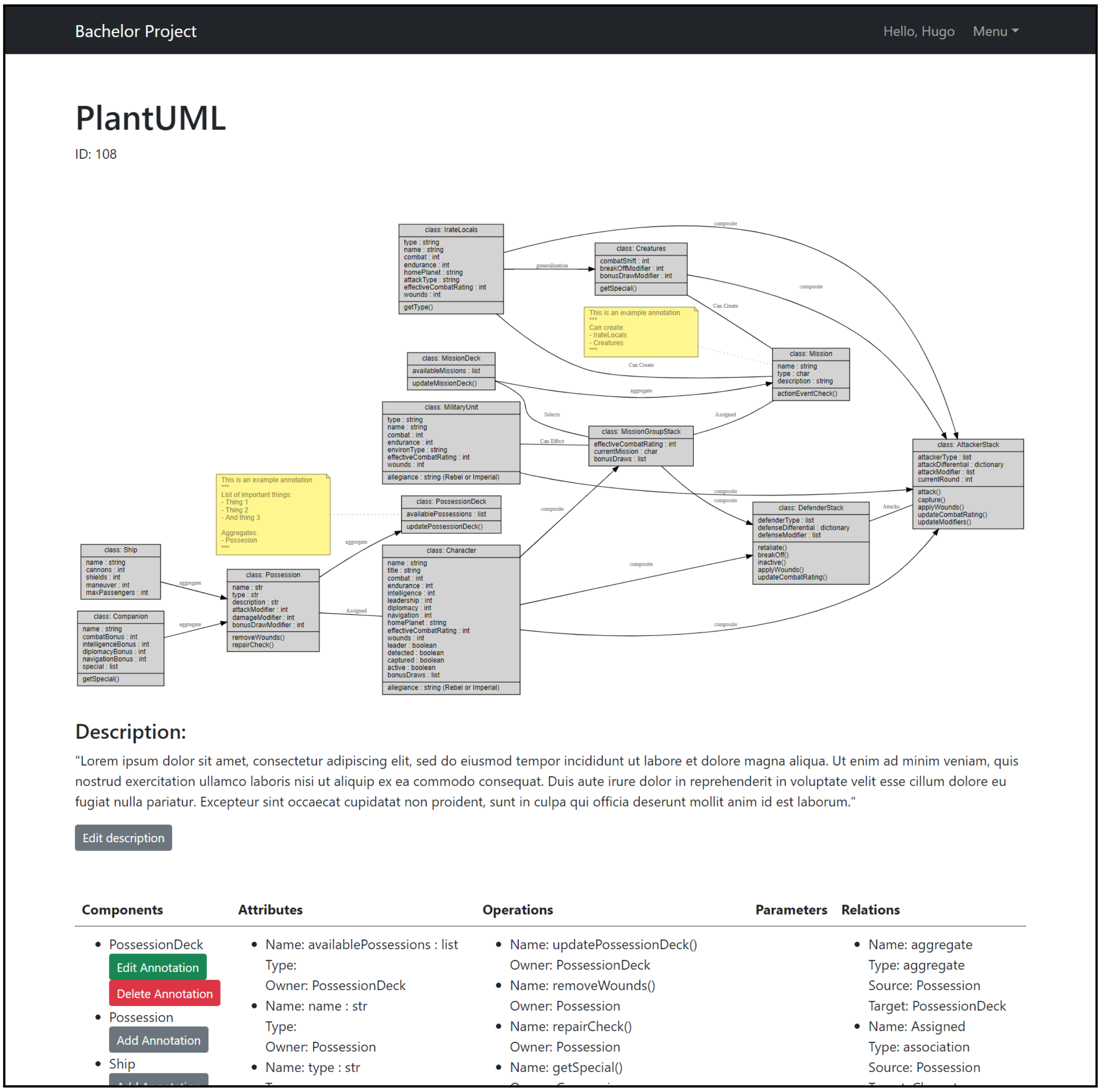
Index Page:

Model	File
PlantUML	relations
PlantUML	components
PlantUML	sample1
PlantUML	sample2
PlantUML	sample3
PlantUML	sample4
PlantUML	sample5

All the models currently in the database.

Figure 11: The Index page: This is the page where the models of the currently uploaded files are listed. A log in is required to see the models. The links take the user to the particular model's detail page.

5.2 Detail Page



PlantUML

ID: 108

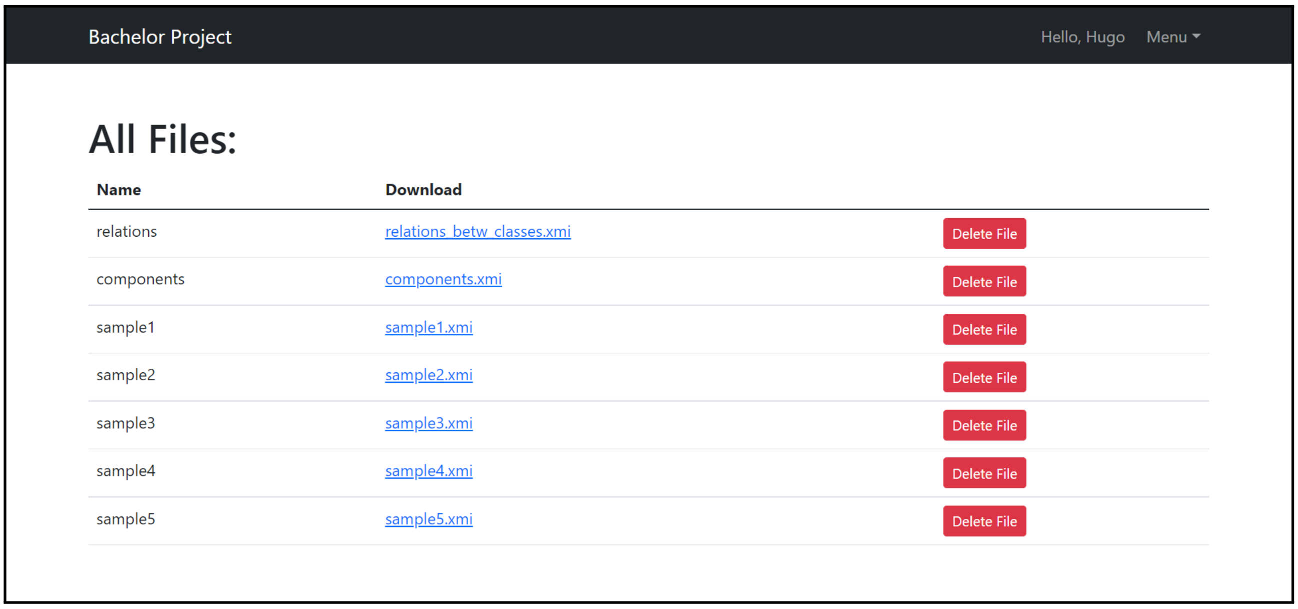
Description:

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

Components	Attributes	Operations	Parameters	Relations
• PossessionDeck Edit Annotation Delete Annotation • Possession Add Annotation • Ship	• Name: availablePossessions : list Type: Owner: PossessionDeck • Name: name : str Type: Owner: Possession • Name: type : str	• Name: updatePossessionDeck() Owner: PossessionDeck • Name: removeWounds() Owner: Possession • Name: repairCheck() Owner: Possession • Name: getSpecial()		• Name: aggregate Type: aggregate Source: Possession Target: PossessionDeck • Name: Assigned Type: association Source: Possession

Figure 12: The Detail page: This is the page where the model is shown along with its details. Shown in the figure is a sample PlantUML Class Diagram model [28] mapped to the general representation. The annotations were manually added.

5.3 File Pages



All Files:

Name	Download	
relations	relations_betw_classes.xmi	Delete File
components	components.xmi	Delete File
sample1	sample1.xmi	Delete File
sample2	sample2.xmi	Delete File
sample3	sample3.xmi	Delete File
sample4	sample4.xmi	Delete File
sample5	sample5.xmi	Delete File

Figure 13: The Files Page: This page shows all files currently in the system. The file is stored locally with a modelfile model instance linked to the location. When the delete button is pressed, the file also gets deleted locally.

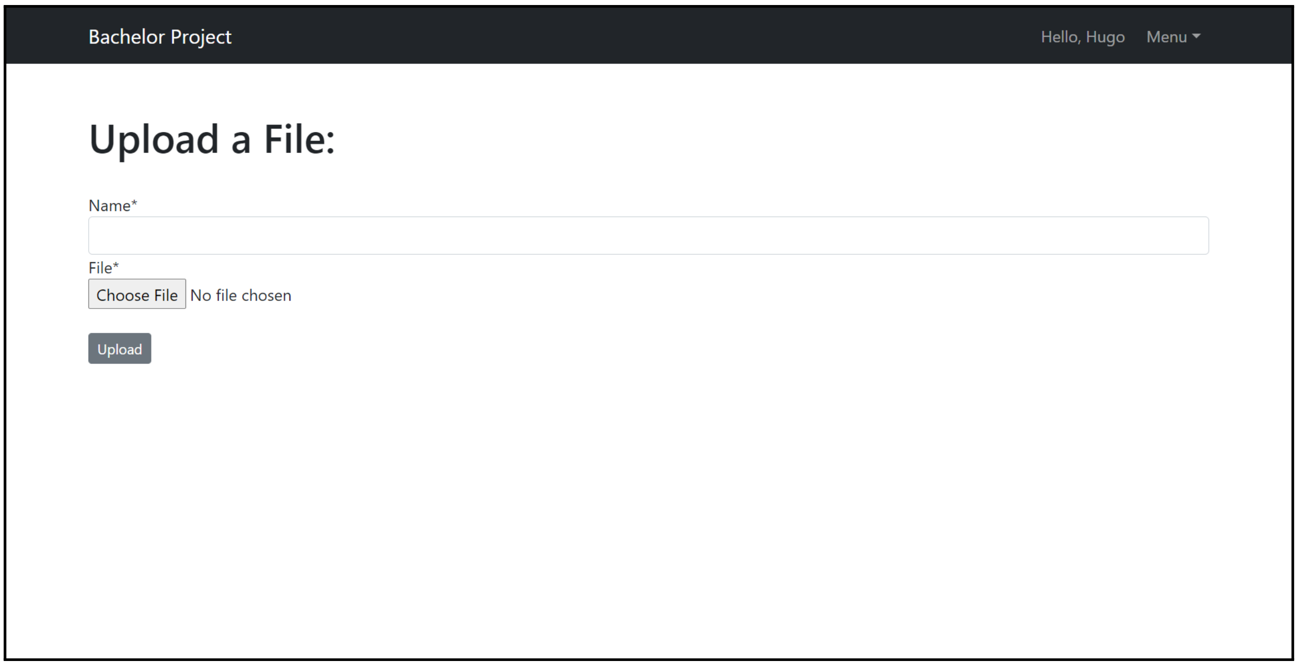


Figure 14: The File Upload Page: Upload a file and assign a name. The file gets saved locally with a modelfile object linked to the file location.

6 Evaluation

The introduced web application offers functionality to import XML-type files generated from PlantUML and StarUML class-diagram models. These models are then visualized in a general representation. This general representation reduces the supported ALs to a graph of components with relations. Descriptions of models can be added, as well as annotations to specific components. The annotations are added to the graph similarly to components, but with an easy-to-differentiate appearance.

The remainder of this section is as follows: First, we evaluate the import functionality. Subsequently, we evaluate the visualization of the general representation and its data model. Then, the usability is evaluated. Finally, we discuss how the application can be extended.

6.1 Datamodel and Visualization

The generalized datamodel consists of components, that are connected with relations. Component and Relation have strings for a type, and name. This allows for the inclusivity of all types of components to be mapped to the general model. If a component type is specified, it is printed before the name in the visualization. Similarly for a relation, the kind of relation is shown as a label, and whether to print a directional edge or non-directional is defined by type.

Attribute and Operation are defined in the datamodel to not lose essential information about the components. Attribute consists of the information that is owned by the component, and Operation contains the actions the operation can perform.

The visualization of the model is kept as simple as possible while keeping the information given clear. The colors are chosen to be easy on the eyes while remaining easy to read. The graph's structure is from left to right, meaning a relation's source component is always more to the left than the target component unless there is circular relation present.

6.2 XML Import

The way XML files can be parsed and processed works well and can be done with the ElementTree Python API. However, the functionality to import using XML-based files brings some issues. Some ALs don't support XML export and use different files to exchange model data between tools or users. For these ALs the import has to be done differently. One way to resolve this issue is to develop a parser for these specific files. This can be done but is currently beyond the scope of this work. Fortunately, XML is widely used, and the 2 most used formal languages found in [1] use a standard model exchange format specified in XML.

Another concern using XML files is when there is no standard XML schema for formatting the model data. This is the case for the UML model export of PlantUML and StarUML. These UML diagram editors both use different XML formatting. This is not a huge issue, but we found PlantUML to not specify the complete required data in the XML file. For example, when using additional arrows in the label of a relation in a class diagram. These additional arrows are another way to specify the direction of the relation but are not included in the XML data.

6.3 Usability

The web application can be easily accessed through any browser. Security is provided by requiring users to register and log in before getting access to the models currently in the database. The models are currently not user-bound, but this can be added in the future. More security can be added by only allowing certain email addresses to register in combination with account activation through email.

Once logged in, new files can easily be uploaded, processed, and deleted due to an easy-to-navigate menu. Once processed, the models will show up on the homepage of the application.

6.4 Extending the Application

An important part of the application is to allow support for the import of models generated by multiple ALs. As it currently stands, only UML class-diagram models from two editors are supported. The way this can be realized is by writing functions that process the tree of the XML file. These functions will have to be specific for each XML formatting type. In other words, every AL will use a different XML format so specific functions will have to be written to explore the distinct trees.

Often in ALs, different types of annotations are supported. this should also be considered for the application. Different colors can be associated with different types of annotations. We now suggest different types of annotations. The annotation types should be general enough to apply to different domain-specific architectural languages while keeping the number of types to a minimum.

We suggest the following types of annotations:

- Informative Annotations: This type of annotation is a basic type of annotation that should be used to add any general information to a part of the model. By defining this type we aim to add support to explain parts of the model to make communication easier.
- Developmental Annotations: The developmental annotation adds information on the state of development of parts of the model. An example is if a component in the model has already been developed or if it is only a concept stage.
- Management Annotation: This annotation type describes what is required for the management of part of the system. This could be contact information of the people responsible, or the resources necessary for the development or deployment of parts of the system.

Another way to extend annotations is by allowing users to define annotation types themselves. This can be done by letting the user choose which shape and color are chosen when creating a new annotation. Using this functionality, the user can create an annotation standard, with specific colors and shapes representing specific annotation types.

7 Conclusion

In this thesis, the deficiencies of ALs are identified. One of these deficiencies is found to be insufficient support for communication. To extend ALs to offer more support for communication a generalized data model is defined and functionality for mapping to this data model is provided. This way models generated by different ALs are represented in a general representation. The general representation consists of components with relations and describes the model in a very simple and easy-to-understand way. This allows users to understand models from different ALs without the need to be trained to understand these specific ALs. Additionally, annotations and a description can be added to the general representation, adding further support for the communication of the model.

The need for communication in the development of the architecture of software can especially be observed in software architectures with very different systems embedded into a larger system. According to Reuter and Andrikopoulos [29], one of these systems, of which development is a challenging task due to the wide variety of stakeholders and engineers from different disciplines, is a Cloud-enabled Cyber-Physical System (CCPS). A CCPS is a Cyber-Physical System(CPS), which means it is a physical and engineered system whose operations are monitored, coordinated, controlled and integrated by a computing and communication core[30]. This core component is linked with cloud systems in a CCPS. CPSs are often limited in storage and computational abilities due to size concerns and being embedded into bigger systems. One example of CPSs includes wireless sensor networks, which record and organize this data to then send it to a central node. Some other examples are autonomous automotive systems, medical monitoring, distributed robotics, and automatic pilot avionics. The difficult design choices in the development of the architecture of a CCPS include what components should be placed on the device, in the cloud, or even in a sub-device on or near the main device[8].

UML class diagrams generated by the PlantUML or StarUML editors are supported for import into the application. The mapping from AADL and Archimate, the two most used formal ALs according to [1], is provided as well. The application is designed to be extensible by adding import functionality for additional ALs.

Annotations can also be extended on by defining specific annotation types or allowing for user-defined annotation types. The three suggested pre-defined annotation types are informative, developmental, and management annotation. The goal of these three suggested types is to cover all necessary information that needs to be added to models of different specific ALs.

The application's functionality is partly based on the ability to map multiple ALs to the new representation. Currently, UML is supported. It is a priority to add more ALs. Fortunately, developing the functions used for this is straightforward. Potentially, this allows users to define the functions for a specific AL themselves.

Bibliography

[1] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, “What industry needs from architectural languages: A survey,” vol. 39, p. 869–891, Jun 2013.

[2] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, “Viewpoints: A framework for integrating multiple perspectives in system development,” vol. 2, p. 31–57, Mar 1992.

[3] G. Kotonya and I. Sommerville, “viewpoints for requirements definition,” vol. 7, no. 6, p. 375, 1992.

[4] “IEEE Recommended Practice for Architectural Description for Software-Intensive Systems,” *IEEE Std 1471-2000*, pp. 1–30, 2000.

[5] “Iso/iec/ieee systems and software engineering – architecture description,” *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pp. 1–46, 2011.

[6] E. Woods, “Architecture description languages and information systems architects: Never the twain shall meet,” 2005.

[7] N. Medvidovic, E. M. Dashofy, and R. N. Taylor, “Moving architectural description from under the technology lamppost,” vol. 49, no. 1, p. 12–31, 2007.

[8] D. G. Kompanek and A., “Reconciling the needs of architectural description with object-modeling notations,” 2000.

[9] E. Dashofy, A. van der Hoek, and R. Taylor, “An infrastructure for the rapid development of xml-based architecture description languages,” ICSE ’02, p. 266–276, ACM, May 2002.

[10] N. Medvidovic and R. Taylor, “A classification and comparison framework for software architecture description languages,” vol. 26, p. 70–93, Jan 2000.

[11] E. Woods and R. Hilliard, “Architecture description languages in practice session report,” 5th Working IEEE/IFIP Conference on Software Architecture (WICSA’05), p. 243–246, IEEE, 2005.

[12] PlantUML, “Class diagram,” *taken from: https://plantuml.com/class-diagram*.

[13] iso.org, “Unified modeling language (uml) version 1.4.2,” *ISO/IEC 19501:2005 - Information technology - Open Distributed Processing*, pp. 1–46, 2011.

[14] R. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, “Model-driven development using uml 2.0: promises and pitfalls,” *Computer*, vol. 39, no. 2, pp. 59–66, 2006.

[15] A. Mehmood and D. Jawawi, “Aspect-oriented modeling approaches and aspect code generation,” *International Review on Computers and Software (IRECOS)*, vol. 8, pp. 983–995, 04 2013.

[16] P. Feiler, D. Gluch, and J. Hudak, “The architecture analysis design language (aadl): An introduction,” Tech. Rep. CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006.

[17] P. H. Feiler, B. A. Lewis, and S. Vestal, "The sae architecture analysis design language (aadl) a standard for engineering performance critical systems," in *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pp. 1206–1211, 2006.

[18] J. Delange and P. Feiler, "Architecture fault modeling with the aadl error-model annex," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 361–368, 2014.

[19] P. Dissaux, J.-p. Bodeveix, M. Filali, P. Gaufillet, and F. Vernadat, "Aadl behavioural annex," 07 2006.

[20] The Open Group, "Archimate 3.1 specification: Chapter 6," *available at https://pubs.opengroup.org/architecture/archimate3-doc/chap06.html*, 2019.

[21] The Open Group, "Archimate 3.1 specification: Chapter 7," *available at https://pubs.opengroup.org/architecture/archimate3-doc/chap07.html*, 2019.

[22] The Open Group, "Archimate 3.1 specification: Chapter 4," *available at https://pubs.opengroup.org/architecture/archimate3-doc/chap04.html*, 2019.

[23] L. Qiuyan, T. Jie, P. Qiuhong, W. Ji, and L. Chao, "Automatic transformation technology from aadl model to uml model," in *2011 IEEE 3rd International Conference on Communication Software and Networks*, pp. 255–258, 2011.

[24] Institut Supérieur de l'Aéronautique et de l'Espace, "The architecture analysis and design language: an overview," *available at": http://www.openaadl.org/downloads/tutorial_models15/part1_introducing_aadl.pdf.*

[25] The Open Group, "Archimate 3.1 specification: Chapter 5," *available at: https://pubs.opengroup.org/architecture/archimate3-doc/chap05.html*, 2019.

[26] MDN contributers, "Django introduction," *available at: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction*, 2021.

[27] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (xml) 1.0 (fifth edition)," *W3C Recommendation, available at http://www.w3.org/TR/REC-xml/*, 2008.

[28] S. Harris and R. Adams, "Character combat and dependencies," *taken from: https://real-world-plantuml.com/umls/4847661404389376.*

[29] A. Reuter and V. Andrikopoulos, "Architectures of cloud-enabled cyber physical systems - a systematic mapping study," 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), p. 455–462, IEEE, Aug 2020.

[30] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Design Automation Conference*, pp. 731–736, 2010.

Appendices

A Examples

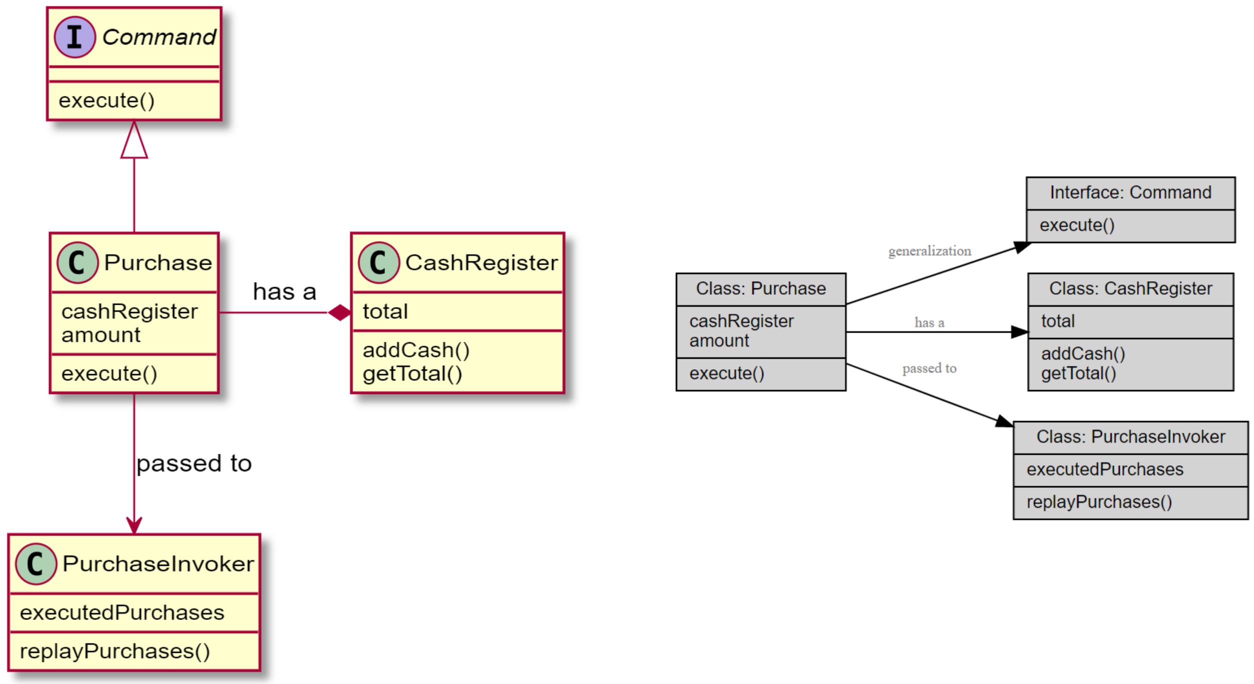


Figure 15: A very simple example of a UML class diagram model converted to the general representation. Most information associated with the components is retained.