Ulm University | 89069 Ulm | Germany

Faculty of Engineering and Computer Science
Institute of Databases and Information Systems

Designing and Implementing a Modeling and Verification Tool for Object-aware Processes

Master's Thesis at Ulm University

Submitted By:
Florian Rotter
florian.rotter@uni-ulm.de

Reviewer:
Manfred Reichert
Rüdiger Pryss

Supervisor:
Kevin Andrews

2016

Version October 27, 2016

Satz: PDF-LaTeX 2_ε

Abstract

This master's thesis focuses on the modeling component of PHILharmonicFlows. This component had to be reworked based on the Universal Windows Platform. Therefore the thesis explains the design and the implementation itself. The rework was needed to ensure that the framework can be used in the future and also to be able to use the modeling tool on different devices. The design focuses especially on the expressions and the roles and permissions. While all points did exist in previous versions they were not satisfying for the user. Therefore the original concept was adapted and extended.

For the expressions a new concept was created that allows the modeler to create expressions that are understandable by the system. While the expression-modeling is completely free it is guided at the same time and errors are automatically prevented whenever this is possible.

In case of the roles and permissions the old concept allowed the user to see and adapt them in a table. This table was not very intuitive and the user had to invest much time to understand the whole concept. This was simplified in the course of this thesis. The goal was to give the modeler an understandable way to set and adapt roles as well as permissions. To achieve this goal the input of both was shifted from the table to the single objects they belong to. This way the user can adapt them better.

Furthermore the verification process has been analyzed and an example of how the verification can be done and a way for errors to be reported to the user was built to demonstrate the applicability. It is important to note that although there is only a single verification running at the moment the concept is built so that it can be extended in the future without any problems.

Acknowledgments

This master's thesis was completed when the author was studying at Ulm University. The author is grateful for the support he received from the Institute of Databases and Information Systems for the duration of the thesis. Especially he wants to thank his supervisor Kevin Andrews and his college Sebastian Steinau for always being open minded and granting their personal support whenever it was needed.

Furthermore the author wants to thank all the people that took their time in order to proofread this thesis.

Contents

1 Introduction

1.1. Motivation

Globalization changes the economy because processes of enterprises become more and more complex. In order to face this change, enterprises have to adopt their strategy and use Business Process Management Systems in order to guarantee a trouble-free process that they are able to improve constantly in order to get more efficient ([27]). With the globalization managers face many problems because "Business Process Management presents them with difficulties from the outset because of the question of language and meaning" ([6]). They have to decide which modeling language they want to use and which parts of their organizational processes should be modeled in order to stay successful.

In reality, nearly all available tools like ERP-, CRM or SCM-Systems do not meet the requirements for data-oriented Business Process Management Systems because they are not able to allow the reading and writing of information independent from the process at any time. Furthermore, in conventional systems it is often not possible to use different levels of granularity in order to model a process. Also if the modeling is activity-based it cannot be ensured that those activities are worked on in the correct order because the system cannot check that ([12, 13, 14]).

In order to meet the described problems, PHILharmonicFlows, a data-centered Business Process Management System, was developed at Ulm University. This system should be able to allow process modeling in a modeling part as well as process execution in a separate run-time tool. "The term business process modeling is used to characterize the identification and (typically rather informal) specification of the business processes at hand. This phase includes modeling of activities and their causal and temporal relationships as well as specific business rules that process executions have to comply with" ([19]). Because of this, the modeling is a very important part and has to be done within the application itself to ensure at least syntactical correctness rather than by some external source.

1.2. Goal of this Thesis

The goal of this master's thesis is to redesign and re-implement the modeling tool of PHILharmonicFlows based on a modern framework with guaranteed future value. Although the modeling component was already created years before ([1]) an update was necessary because the overall goal is to allow schema evolution and building of variants in one single tool. Furthermore, it should be possible to use the same framework for the modeling and the run-time part of the application. This way the user experience is better. It is also important to note that the updated version should be able to run on different kinds of devices, which was not possible for the earlier versions.

Besides the above mentioned goals it was also necessary to update single components of the earlier version to improve their usability and allow more flexibility and guidance for the modeler. Those parts are listed beneath.

- Modeling of Expressions
- Role System
- Permission System
- Verification of modeled parts

Although some parts were fully functional those parts were not state-of-the-art technology because they did either not use the full spectrum modern systems can provide or were not designed in a satisfying way for users. Moreover, in case of the verification there was never a working implementation that could be used to demonstrate how a generic model would be designed and used.

The thesis addresses all of the points mentioned and creates a basis for the further development of PHILharmonicFlows.

1.3. Outline

The master thesis is structured as follows. In Section 2 there is an overview of fundamental parts of the PHILharmonicFlows-Application. This covers an insight into the original concept as well as a short statement regarding the technologies that were used to build the application itself. The fundamentals are followed by a general explanation of the application in Section 3. Included in this section are all available process graphs as well as the graphical objects that are used to build it and a short explanation about the usage of all of them. After this more superficial section, the thesis gets more detailed. The key features of the application are explained in Section 4 to 6. The term "key feature" in this context means that it is an integral part of the application without which the application could not work and that the implementation was very difficult and time consuming compared to other parts.

The first key feature is the Expression Modeler which is described in detail in Section 4. Followed by an insight on how roles and permissions are handled by the application in Section 5. The last key feature is described in Section 6, which is about process verification. This section does not only include an analysis of properties that have to be verified but it also includes a working example that shows how the verification is done by the system and how the errors can be reported to the user. The important parts are also depicted in Figure 1.1.

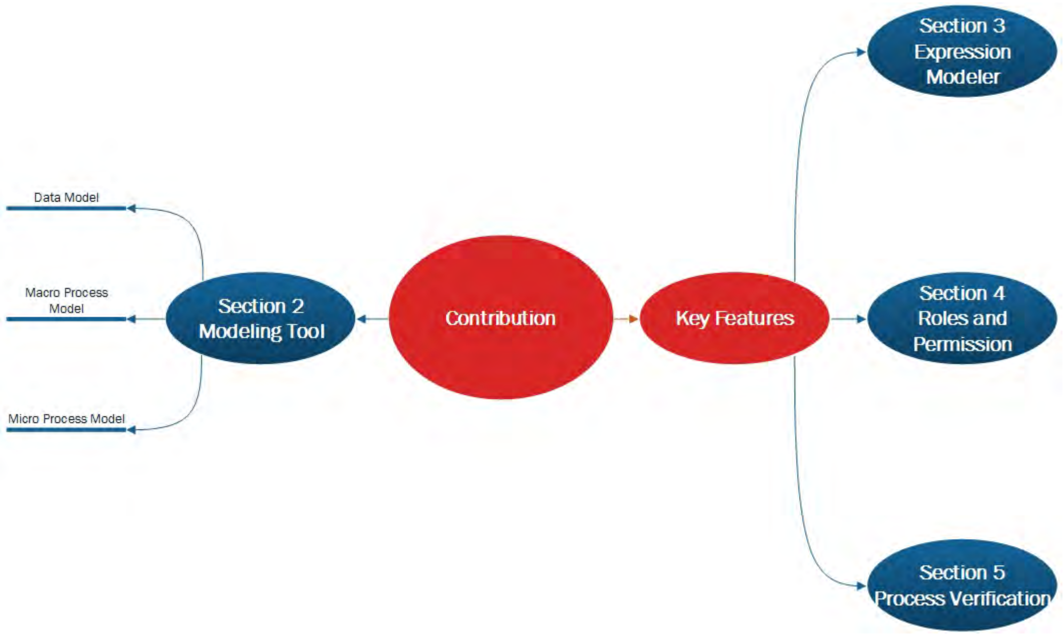


Figure 1.1.: Contribution of the thesis

In the last two sections of this thesis there is first an overview of related work that is done by other scientists in the field of process management (Section 7) and secondly a short summary followed by an outlook for PHILharmonicFlows and the possible future of this application (Section 8).

2 Fundamentals

This section is used to explain theoretical and technical fundamentals of the thesis. The concept of PHILharmonicFlows will be explained in detail (Section 2.1), followed by an analysis of the requirements in Section **??**. There is an overview of the technologies that were used to design and implement the modeling tool, which is the main concern of this thesis, in Section 2.3.

2.1. PHILharmonicFlows

In general PHILharmonicFlows is a framework for object-aware processes and therefore possesses the following major characteristics which were described by Vera Künzle and can be seen in Table 2.1.

Characteristic	Explanation
Object behavior	The behavior of the involved business objects must be taken into account during process execution.
Object interactions	Interactions between business objects must be considered.
Data-driven execution	Process execution has to be accomplished in a data-driven manner.
Integrated access	Authorized users must be able to access and manage process-related objects at any point in time.
Flexible activity execution	Activities must be executable at different levels of granularity.

Table 2.1.: Major characteristics for object-aware processes ([10, 17, 16])

PHILharmonicFlows is an acronym that stands for "Process, Humans and Information Linkage for harmonic Business Flows". The basic framework was introduced in the year 2012 and since then continuously developed and improved. Whereas the technical parts are now mostly fully functional and run on the universal windows platform, the basic concept still stayed the same. The framework is composed of two parts. First the modeling part, that is were this thesis is placed, and second the run-time part, that is developed on its own. Both parts share the same data that is provided by a server and already interact with one another. Figure 2.1 illustrates the complete framework. The different parts of the modeling tool are explained in more detail in this subsection.

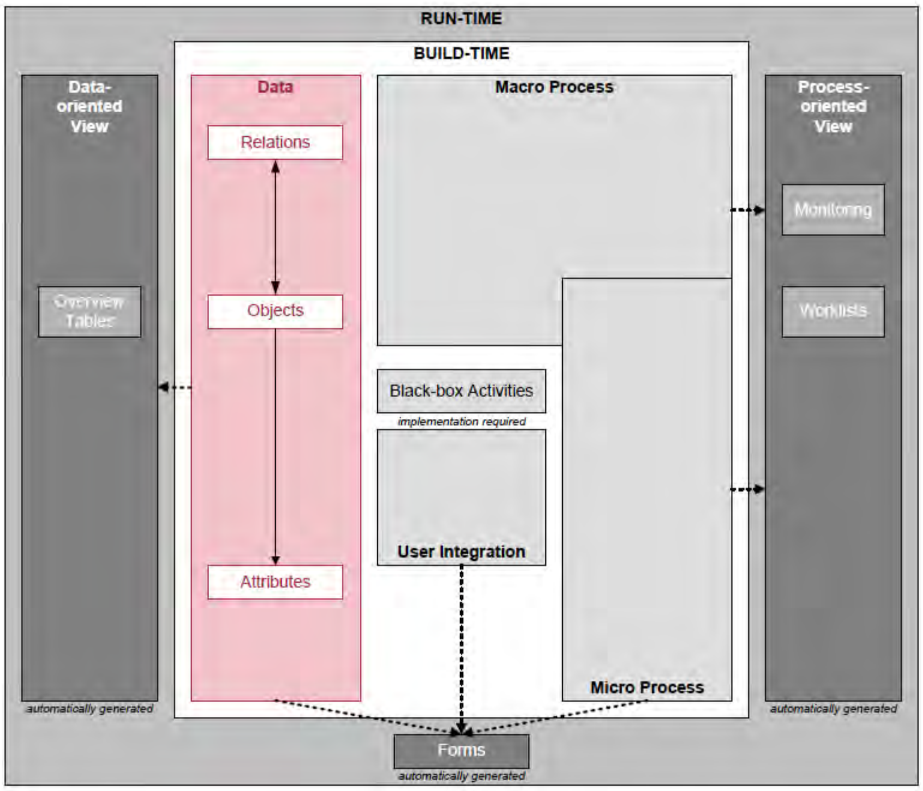


Figure 2.1.: Overview of PHILharmonicFlows ([11])

2.1.1. Data Model

The framework relies on a relational data-model and therefore consists of different types of objects (object type) with specific attributes (attribute type) and particular relations (relation type) between them. Additionally PHILharmonicFlows is a data-driven Process Management System (PrMS). The first step for a user when he starts working with the system, is to specify and model a data model as the base for all other process parts. An object type can be seen as class and can be identified by an object identifier. This identifier is not a primary key as it is normally necessary due to the restrictions of a relational database but depends on the technical implementation. In the case of the application that is described in this thesis the object identifiers are numeric values that are generated automatically by the actor framework, explained in Section 2.3.

The attributes of each object type are atomic values and can be seen as the properties of an object type. Each such attributes comprises a name, a data type and a value ([11]).

The specific data types and their implementation are explained in Section 3.2. Not every attribute might be needed in every instance in the run-time. So attributes can be required or optional, this is specified in the micro process. Also it is possible to assign a default value to an attribute. This is used for example with attributes that have a numeric value, in this case the default value is normally zero. Also it is possible to predefine a set of values for an attribute. This is used to support the user in the run-time because he then can only choose from the given options and also to prevent him to insert wrong values like typos or values that are wrong due to the context. It is important to note that an attribute type can not only be a classical data type like numeric or boolean but it is also possible to set it as a reference to other object types.

Between different object types the relation types are used to model semantic relationships. To each relation type a minimum and a maximum cardinality can be assigned. This way relations can be made more specific. However, as it is usual in relational databases, each many-to-many relation has to be dissolved by using an additional object so that only one-to-many relations are in the data model. As a reason for the dissolving there are transitive relations in the data model because every dissolved many-to-many relation automatically generates one transitive relation due to the inserted object ([11]). Furthermore, it is important to note that the data model has to be acyclic.

At run-time, a variety of instances of an object type can be created. Each of those instances usually has different attribute values as well as relations. The number of those instances has to meet the specified cardinality.

Process management and data integration can be done by a data- and/or task-oriented view. Both views should not be treated completely independent from each other, because if, for example, data is accessed the corresponding process state is important as well ([11]). To serve this purpose, the application has some overview tables for the user that are generated automatically. In these tables all column descriptions are from the attribute types of the objects that are referred by the relations. The description itself is from a label attribute that every object has and that allows naming each object that is used in a readable way for the user. This is mandatory for better understanding and handling the application.

2.1.2. Micro Process

The micro process is used to model object-behavior and has to be defined for every object type from the data model. At the run-time, for each created instance of an object, there will be a micro process instance as well. Also, a micro process allows the process modeler to use elements from the imperative as well as the declarative modeling paradigm. Furthermore “PHILharmonicFlows enforces a well-defined modeling methodology that governs the process engineer to define processes at different levels of granularity” ([11]). The finest of those granularities is defined with the use of micro step types and value step types. A coarser granularity is defined with the help of micro state types.

Within the micro process the user can handle mandatory, as well as optional data for the process. Mandatory data has to be set before the process can move to the next state. The modeling allows the specification in which order the specific data has to be set in the run-time. The process comprises different micro step types that are connected to one another by micro transition types. For each data input that is mandatory, there exists a micro step type, because a micro step is used to refer the attribute types from the data model. It is possible that a micro step does not refer to any attribute type, this is the case for start and end step types. They are called empty micro step types. An empty step is immediately reached when one of the incoming transitions is activated. A micro step type that refers to an attribute is called an atomic micro step type and it is reached when the value of the referred attribute is set in the run-time, if the value is still missing a work-list notes this specific attribute as mandatory to set. The last type of micro step is called value-specific micro step type which means that the micro step type contains at least one value step type ([15, 18]).

A value step type can only be placed within a micro step type and is used to express, that an attribute can depend on values of other object attributes besides the fact that it has to have its own value set. Therefore, different values or value ranges can be modeled with the value step types. The constraint that is represented by the value step type or more precisely the expression that is part of the value step type can be validated and will become activated if it is satisfied. Each value step type can be the source of a micro transition type.

As discussed, micro transition types are used to connect either micro step types with another micro step or value step types with a micro step type. Because of that, they define the order in which attributes values and relations shall be set at the run-time. Of course, flexibility is not restricted by this design because it is possible for the user to insert specific values before they are needed and if the process reaches a particular state that requires the already written data, it will acknowledge the fact that the date already exits. This way, data-driven execution is supported. It is also possible to change the value of attributes at any given time if this is necessary. As noted earlier, the source of a micro step can also be a value step type. This opens different execution paths for the process. However, it has to be ensured that only one path can be chosen at the same time. That is the reason why micro transition types originating from the same source get different priorities. In case that more than one relation might become activated, the system evaluates the priorities and chooses the one with the highest priority. To enable this of course the priorities that can be assigned to the relation types have to be distinct. The prioritization is also necessary if the micro transitions originating from different value steps that are part of the same micro step type. If more than one value step could be fired, the one with the micro transition which has the highest priority will be activated, the others will not. To prevent deadlocks, a micro transition originating from a micro step type that contains value step types is possible. In this special case the micro transition from the micro step will be seen as default transition and fired if no value step can be activated.

While the fine grained modeling is done with the help of micro step types, less fine granular modeling can be done with state types. Those are used to coordinate the order in which attribute values have to been set by different users. Furthermore, the modeling of black-box activities is supported at this granularity level. Black-box activities are specified in advance and cannot be changed by the user. One example for this, is the process step to “send an e-mail”. This can be implemented as a black-box activity, because the only important inputs for the modeler are sender, receiver and content of the mail. The implementation of the specific mail client or other technical details do not concern the user and can be handled by different persons, like a system administrator.

State types “are used to realize mandatory activities and coordinate the processing of individual object instances among different users” ([11]). Therefore, a state type comprises one or many micro step types, each representing an attribute value from the data-model, that has to be set within a form at run-time. Each state can be associated with a permission, to execute it, which implies that he is also associated with a particular role that possesses the permission. In this context, it is also necessary to mention that a micro transition type that connects micro step types of different states is a “external” micro transition type while the connection of two micro steps within a state is called “internal”. Because the state can only left and therefore executed with external micro transitions each of those holds a state execution permission.

Another form of edge that exists between different states is the backward transition type. This particular transition allows the user to do backward jumps within the process if he possesses the so called backward jump permission. Those backward jumps can be necessary in some scenarios, especially to get flexibility for the user in the case that one of the external conditions of the process has been changed. An example for that can be found in an application process. In case an applicant that would have been approved cancels his application, it is necessary that the caseworker is able to change not only this particular process but maybe also other processes that were executed due to this particular job offer. It is important to note that a backward jump can only be executed between a successor state and a state that is in the subset of its predecessors.

Due to the different facts that were discussed it should be obvious that a real data-model that comprises many different object and user types would need a lot of micro processes. To reduce administrative efforts, PHILharmonicFlows enables the generation of minimal micro process types for an object as well as a user type. This process always is composed of a start and an end state.

2.1.3. Macro Process

While the micro process allows the modeling of the process within an object type the macro process is used to model interactions between different object or user types. Furthermore, a macro process “allow[s] hiding the complexity of large process structures form users” ([11]).

PHILharmonicFlows automatically creates a state-based view for a micro process, those views are used to define how objects in different states will interact with one-another. This way, many process instances that might be evolved during run-time can be still synchronized at fixed points. Moreover, while synchronizing a large number of micro process instances would be impossible for the user, using abstract and flat macro process types to do this is easier for the modeler. Each macro process type is based on an object or user type and a particular state of those types as it is modeled in the specific micro process of the object or user type. A macro process type comprises macro step types that are connected with macro transition types. Additionally, a macro process can have parallel or alternative execution paths. For that reason, each macro step type is composed of at least one macro port type. Those port types are the target for the connecting macro transition type if different transitions target different ports of the same macro step type an “OR” is modeled if they have the same target an “AND” semantics is modeled because a macro step type will become activated, if at least one of its ports becomes activated.

Besides direct dependencies the macro process is used to show other forms of dependencies too, in order to allow for a more sophisticated process coordination. In this context, dependencies can also be transverse (existing between two object types with the same higher level object-type), bottom-up (relation between upper and lower level object types at which the transition source is the lower level and the transition target the upper level) and top-down (bottom-up but with reverse source and target).

Because it is possible to model a macro process with dependencies from different object or user types, that might also have different micro processes, there is the potential to model a dead-lock, that is the reason why each macro step type also has to refer to a specific state type, to prevent those dead-locks.

2.2. Requirements

This section is used to explain the requirements for the data modeling tool. First, in Section 2.2.1the non-functional requirements are explained, followed by the functional requirements in Section 2.2.2 and last an overview in Section 2.2.3.

2.2.1. Non-Functional Requirements

All non-functional requirements are general requirements and therefore almost exclusively satisfied due to the choice of the development framework (UWP). Nonetheless, those requirements are important and therefore explained in this section. After the explanation there is a short formal requirement formulated.

Scalability

The application should be scalable, so bigger companies with many employees are able to use it. This does especially concern the workload the server is able to manage. Due to UWP this is not a problem, because the Actors (cf. Section 2.3.3), used together with the UWP, allow the scaling of the server.

Therefore the following requirement can be formulated.

Requirement REQ-1: (Scalability)
The application should be scalable.

Tracability

“Requirements traceability refers to the ability to define, capture and follow the traces left by requirements on other elements of the software development environment and the trace left by those elements on requirements” ([26]).

This leads to the following formal requirement.

Requirement REQ-2: (Tracability)
All requirements should be traceable.

Reusability

Each part of the modeling tool should be designed in order to be reusable in other parts of PHILharmonicFlows or, if possible, as a stand-alone asset. This does especially concern the modeling of the graph in general, the expression modeler, as well as the roles and permissions. The first point is important, because the graphs that can be modeled in the modeling tool are at least partially the same graphs used in at run-time. This is also true for the second point. The expression modeler is not only used in the whole application (modeling tool and run-time) but also in every possible graph that can be modeled.

Additionally, it is possible to use the expression modeler alone. The last point, the roles and permissions, are also used in the modeling tool as well as at run-time. Akin to the scalability this requirement is satisfied due to UWP and the development process.

Based on the need to be able to reuse parts of the application the following requirement can be formulated.

Requirement REQ-3: (Reusability)
Each part of the modeling tool should be reusable.

Quality

The experience a modeler has when modeling with the modeling tool has to be satisfactory. This concerns not only the user experience when modeling but also that the application works properly and the tool's performance. This requirement is satisfied due to the development process with constant revision and reworking of the application.

This leads to the following requirement.

Requirement REQ-4: (Quality)
User feeling and application performance should satisfy high standards.

Portability

The application and the server should be able to be portable and therefore used in different environments. This is important due to the fact that one day the application should be able to run not only on different devices, like PC, tablet, mobile phone, but also as a web based version. This, however, needs a web-interface that supports CRUD-operations and can work with any client. As the other non-functional requirements this one is satisfied due to the chosen framework, that enables a web-interface.

The mentioned goal, that the application is able to run free from any hosting systems or device boundaries, can be formulated as stated the following requirement:

Requirement REQ-5: (Portability)
The application should be able to work in various environments.

2.2.2. Functional Requirements

Besides the non-functional requirements, the modeling tool of PHILharmonicFlows has some functional requirements as well, those are illustrated in this section. However, it is important to note, that all requirements that are necessary only for the run-time part of the application are not part of this overview.

Simple Modeling

The complexity of the PHILharmonicFlows concept makes it necessary to design a modeling tool which is as easy to use as possible, so that the user only has to deal with the model he will create and not the difficulties of the tool. For that reason, the application has to be designed in a way that it is very easy for the user to get a general overview. This includes that as few as possible elements have to be dragged by the user, as well as not overload the user while modeling. Especially for the last part it is necessary to hide unnecessary information from the user so that he can focus on modeling.

This point allows the formulating of the following requirement.

Requirement REQ-6: (Simple Modeling)
The modeling tool has to be designed as easy to use as possible.

Correctness by Construction

One important requirement of the modeling process is that the user can only model correct models whenever this is possible, this includes the need to prevent actions that are obviously wrong as well as the verification of each step on the server. For example, in general the vertices of the models can be dragged to the graph by the user, this allows him to feel more involved with the modeling. However, the user cannot position the dropped vertices wrong or completely free, because the position of the drop will just be ignored or automatically corrected by a placement algorithm. This will also support the requirement for simple modeling (cf. REQ-5) beacause it takes away complexity from to user and hides that within the business logic of the application or the server.

The need of correct models can be formulated as the following requirement.

Requirement REQ-7: (Correctness by Construction)
The user should only be able to model correct models whenever this is possible.

Simple Expression Modeling

Also, an important requirement is that expressions, that are used widely within the application and are explained in detail in Section 4, are as easy as possible to model. Because the application supports many different functions that might not be common for all people it has to ensure that the user cannot make errors that are avoidable.

This leads to the following requirement.

Requirement REQ-8: (Simple Expression Modeling)
Expressions have to be as easy as possible to model.

Simple Roles

Roles “provide integrated access to business data on one hand and business processes on the other require advanced concepts for user integration” ([11]). As it is common in every application with a rune-time part roles are very important to define competences. Within the original concept of PHILharmonicFlows roles were introduced and while they are working in theory the tabular user interface to adapt them was not state-of-the-art. Therefore the modeling tool requires to improve the roles and simplify them for better understanding of the user. This includes a native placement as well as an understandable general concept. Furthermore, it is necessary to allow the input for roles directly in the modeled graph.

The reasons above can be formulated as the following requirement.

Requirement REQ-9: (Simple Roles)
The setting of roles has to be simplified for better understanding by modeler.

Simple Permissions

Akin to roles, permissions were introduced in the original concept of PHILharmonicFlows as well. Due to that, they face the same problems because they belong to roles and therefore used the same tabular input as roles. Because of this, they also have to be reworked. As for the roles, the permissions have to be set directly in the graph. Therefore it is necessary to split them from the roles they did go with in the concept and allow the user to insert them separately. This way, the modeling flow is more natural and the user do not have to use a confusing table.

Similar to the roles, the following requirement is based on the mentioned reasons.

Requirement REQ-10: (Simple Permissions)
The setting of permissions has to be simplified for better understanding by modeler.

Simple Verification Concept

As every framework with a run-time part, PHILharmonicFlows has the need to verify the modeled graphs to prevent the server from crashing. Besides that, a verification concept has to come with additional error reporting, that supports the user and enables him to understand and correct found errors. It is also important to note, that the verification concept is not only limited to correctly modeled graphs but also to all other components. This concerns the used expressions, as well as other inputs, like attributes in the data model or roles and permissions.

This allows to state the following requirement.

Requirement REQ-11: (Simple Verification Concept)
The modeling tool has to have a verification concept that is able to verify the process in general, as well as each part of the modeled process on its own.

2.2.3. Summary

This section gives a short overview of all requirements fulfilled by the application of this master's thesis. Therefore, all requirements, that are stated above, are listed in Table 2.2.

REQ #	REQUIREMENT DESCRIPTION	RESOLVED IN
REQ-1	Scalability	framework
REQ-2	Traceability	development
REQ-3	Reusability	framework & development
REQ-4	Quality	development
REQ-5	Portability	framework
REQ-6	Simple Modeling	Section 3
REQ-7	Correctness by Construction	Section 6
REQ-8	Simple Expression Modeling	Section 4
REQ-9	Simple Roles	Section 5
REQ-10	Simple Permissions	Section 5
REQ-11	Simple Verification Concept	Section 6

Table 2.2.: Overview of Requirements

2.3. Technologies

The technology used in the implementation of the application for this master's thesis was predetermined due to the general vision that one tool should be able to do the modeling as well as the handle the run-time part of PHILharmonicFlows. This is necessary because the overall goal was to allow the evolution of schemes as well as variants with one single tool. Due to the fact that the run-time is even more complex than the modeling, the design and implementation of this part started in an earlier stage of the project and therefor the chosen framework "Universal Windows Platform" is the basis for the modeling part as well.

2.3.1. Universal Windows Platform

When Microsoft released Windows 10 as their new operating system, they introduced the Universal Windows Platform (UWP), "which further evolves the Windows Runtime model and brings it into the Windows 10 unified core" ([24]). Because UWP runs on many different types of devices, like a PC, mobile devices, Surface Hub or holo lenses it is possible to use the application on all of those. Figure 2.2 shows the whole environment of the platform. Using this platform supports one important goal of PHILharmonicFlows, which is that the application should be able to be used on different devices. While mobile phones and gaming consoles might not be primarily targeted because the first one has a screen that is too small to model and the second one does not typically exist in a company, a PC or a tablet are the main target. Also, new technologies like a holo lens or the extra large screen of a Surface Hub that enable modeling and discussion in team meetings might not be very common at the moment but they are targeted so that the application has a guaranteed future.

UWP supports the developer with universal input methods so that almost no additional adaptation is needed to run the application on another device family. Furthermore, it also supports the design of adaptive UI and supports user experience due to the fact that a user is able to use the same application on all his devices.

The supported language options of the platform are C++, C#, Visual Basic and JavaScript and the presentation can be done via XAML, DirectX or HTML. For the application that was built during this master's thesis the chosen language is C# and the graphical elements are done in XAML.

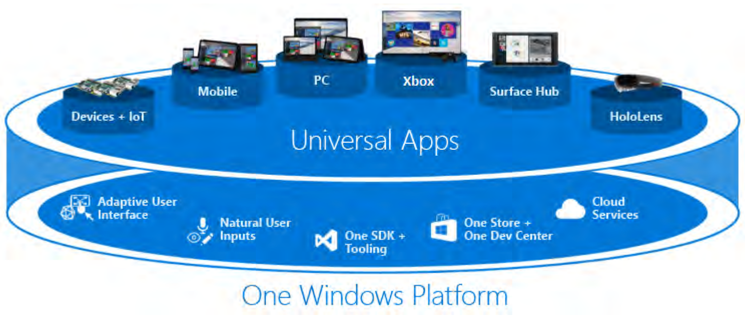


Figure 2.2.: UWP environment ([24])

2.3.2. Windows Communication Foundation

Due to the fact the one goal of the application is that the modeling UI has as little logic as possible all modeling steps have to be sent to a web-service that generates a View Model. This web-service uses the Windows Communication Foundation (WCF), a framework for service-oriented applications. Data can be sent and received asynchronously and also other modeling clients are able to connect to the web-service once it is hosted. This allows the development of other tools that might be more specialized for a particular business. The messages are based on data contracts in which a class is created that represents the data entity. The metadata for this class will be generated automatically so that the UI client is able to use the data types ([22]).

2.3.3. Actors

The server can be placed in the cloud and running in Microsoft Azure with the so called “Service Fabric”. Within the Service Fabric the Reliable Actor Framework is used. This framework is based on the virtual actor pattern designed by researchers in the “eXtreme Compution Group” of Microsoft. The goal of the development was to “encourage the use of simple concurrency patterns that are easy to understand and implement correctly, building on an actor-like model with declarative specification of persistence, replication, and consistency and using lightweight transactions to support the development of a reliable and scalable client cloud software” ([5]).

The actors are used as persistent data storage and therefore a classical database is not needed. An actor is automatically activated the first time it is used and identified by a unique id. To save resources, an actor is removed if it is not called over a long period of time, but the system will remember its state and is able to recreate it. It is also possible to explicitly delete an actor. All actors are stored in the cloud and the framework provides an automatic fail service that ensures that a copy of an actor is existent at every time even if one node of the cloud fails, there will be a copy on another healthy node.

Communication with an actor is based on asynchronous calls on a task basis. Those tasks must be serializable by the system via a data contract. To communicate from “client-to-actor” or “actor-to-actor” a proxy is used. This proxy will locate the actor that is placed randomly in the cloud cluster and open a channel for the communication ([25]).

2.3.4. Coded UI Testing

The PHILharmonicFlows concept is very complex, which is why the implementation is also very difficult. As it is usual in a system, bugs and errors exist and have to be found. While obvious compilation errors can be found relatively fast, semantic errors due to the complex server and data structure need intense testing.

Because testing usually needs manpower which is very costly, Microsoft supports the developers of application for the Universal Windows Platform with coded UI testing. It is based on appium, a test-automation for mobile apps ([30]).

With coded UI it is possible to let the system automatically create a model by following coded instructions and enables finding errors more quickly. Therefore, the application that shall be tested calls a test app that comprises different test scenarios and communicates via a driver [2].

It is planned to implement this feature in PHILharmonicFlows as well to prove that all different kinds of processes can be modeled with the modeling tool, which is also a reason why the application was built with UWP.

3 Modeling

The modeling section describes the core of the application. Its parts are the three graphs that have to be modeled, in order to get a complete model of a PHILharmonicFlows process. General information about the modeling is presented in Section 3.1. Section 3.2 describes how a Data Model is built, followed by the micro process model in Section 3.3 and the macro process model in the last Section 3.4.

3.1. General Idea

The fundamental idea is, that every model is a subclass of *Graph* and therefore it is a grid which is placed within the application window, refer to Figure 3.1 for a brief overview of the single parts of a modeling graph. All parts are numbered to identify them in the explaining.

The super-class *Graph* contains some important parts. First it allows accessing of a so called *GraphGrid* (cf. Fig. 3.1 (3)), this is the *Grid* where the actual objects (cf. Fig. 3.1 (5)) are placed in, second it allows to add and delete objects to or from the *Graph* (edges and vertices) and third it allows to select and deselect single objects of the *Graph*. Moreover it enables the persistent saving of the modeled objects.

Within the *Graph*, there are two types of elements that can be placed. The first element is a *Vertex*, that can be dragged from the side bar (cf. Fig. 3.1 (2)), that is placed next to the option area (cf. Fig. 3.1 (1)). Each *Vertex* can be selected and contains one or more anchors which function as source and target for an *Edge*, the second element. Each edge can be dragged from a source anchor to a target anchor. Furthermore, it can be selected by the user.

Every created view, which is discussed in the following sections, implements either the class *Vertex* or the class *Edge*. Each model has also a *SplitView*. This is a visual object that is composed from two parts. One is an area that is visible all the time. The second is called *Pane* (cf. Fig. 3.1 (4)). Normally it is hidden but it can be opened. If it is opened it will overlap the other things at least partially. In this special area, additional information,in this case attributes (cf. Fig. 3.1 (6)), is served. Within the application, the opening of the *Pane* will be triggered if the user selects an object. This way, not all options have to be visible at any time and the user is not overwhelmed by the amount of information he could get.

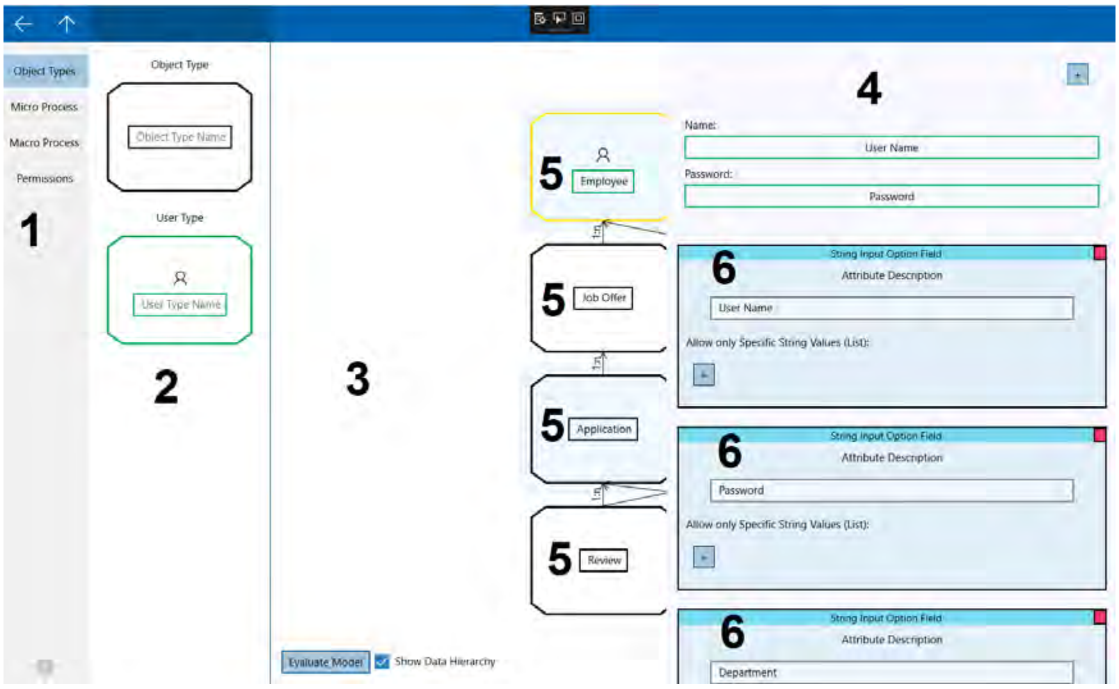


Figure 3.1.: General structure of a modeling graph

It is also important to note that this section (3) illustrates how the above mentioned requirement of “Simple Modeling” (REQ-6) is satisfied. The following sections show, that every model is designed as simple as possible while allowing the user to insert all information necessary. An example for this can be seen in the above mentioned *Pane* (cf. Fig. 3.1 (4)) of the *SplitView*. As it was said, every graph possesses such an area. If the pane is closed the modeling can be done normally, if it is opened all available information is visible for the modeler. This allows the simplification of the modeling process in general due to the fact that only necessary information is visible for the user. It also supports Modeling due to the fact, that the *GraphGrid* (cf. Fig. 3.1 (3)), that is used to draw the model can be significantly larger due to the fact that information can be placed not only side by side but also on top of each other without disturbing the modeling flow.

3.2. Modeling the Data Model

The first step to model a complete process for PHILharmonicFlows is the Data Model. With this model the user is able to specify the different object types and relations between them as well as define attributes that are available for those particular objects.

3.2.1. Vertices

Each vertex is placed on the *GraphGrid*, shown in Figure 3.2. The placement is determined by an algorithm and depends on the relations between the modeled objects. For example, in a newly created graph, with three new vertices, those are all on the same row. If, however, an edge is drawn from the first to the second vertex, then the model is redrawn and the first vertex is on the lowest row while the other two will remain on the highest row. This is also used to show the data hierarchy that is used to depict more sophisticated relations within the model.

The implicit hierarchy can also be visualized, therefore the user can click on a button in the lower left corner of the graph (cf. Appendix B, Fig. B.1 (4)), that will dye every second row in color so that the different hierarchy levels are more obvious. An example for that is also shown in Figure 3.2. The default for this option is to hide it because while in modeling the color of the rows might distract the user.

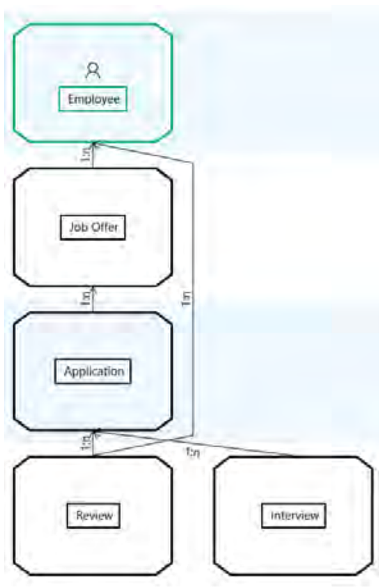


Figure 3.2.: Graph with data hierarchy

Object Type

An object type (cf. Appendix B, Fig. B.1 (1) - for a general overview; Fig.3.3 - for a more detailed view) is an entity that is the basis of the whole application and it can be dragged by the user from the option area into the graph (cf. Appendix B, Fig. B.1 (3)). The whole information of the application is built on objects. The object type entity contains information about the different types of actors that exist for this object. For further reference look at the Section 2.3.3. An object type has a name and can have a list of attributes attached. The information that is used in the modeling tool is served via the web-service in form of a view model. Such a view model has some additional important information for the modeling. First, it has an own id, as well as the ids of the attached micro and macro processes for the object. Obviously the name and the attribute-list have to be displayed in order that the modeler can manipulated them. This is why this information is in the view model as well. Furthermore, there are lists of incoming and outgoing relations, that will be used to draw the actual model. Also there is a list of roles that are available, as well as a flag if the object type is a user type.

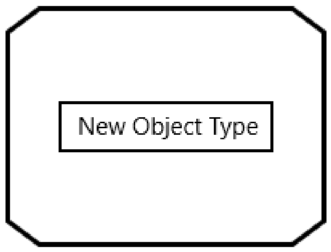


Figure 3.3.: Object type view

For each view model a view can be generated. This view contains all information of the view model because it gets the actual view model via constructor injection. Another method would be to write the view model into the data context of the view. While the actual storage is more or less the same, the injection makes it more obvious and understandable for future code reviews. Also it is possible to prevent the creation of a view if the view model does not exist or is incorrect and the injection is type safe, because of those advantages this method was used for every view.

The object type view follows very simple design. As mentioned, an example can be seen in Figure3.3. It offers only the possibility to insert an object name that has to be unique. However, the object or user type requires that special attributes have to be set. In case of the object type the attribute is the name. It is used to identify which name has to be displayed for each particular instance at run-time. This is necessary so that the run-time can name objects automatically. The attribute can be set after selecting the object type view in the split view, as it is depicted in Figure 3.4. The chosen method ensures that an attribute is selected but gives the user the freedom of choices because he can pick any attribute as long as it is a string attribute. This is the best option because otherwise the application has to create either the name attribute automatically and the user cannot overwrite it or there would be an error, if the user has not created a string attribute and named it “name” or used any other fixed value that could identify the attribute. Also, the free choice in names that comes with attribute referencing supports scenarios in which it is advantageous to have a specific name rather than a generic name. For example, if a specialist is used to the term “description” rather than “name” the person that creates the model could take this into consideration and use the first term so that the specialist will be more familiar and confident in the use of the application. Such a scenario is not supported by a fixed description.

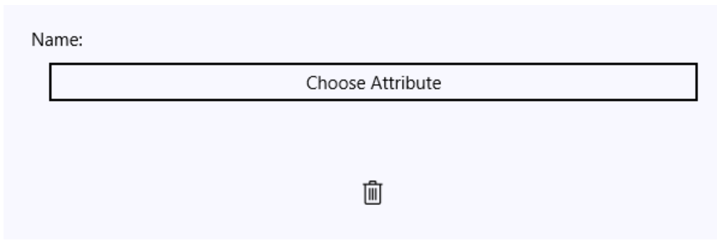


Figure 3.4.: Special attribute for an object type

User Type

The user type is a special version of an object type and can also be dragged from the sidebar (cf. Appendix B, Fig. B.1 (2) - for a general overview; Fig.3.5 - for a more detailed view). On the server it is represented as partial class of the object type entity. This allows a better overview of the class and makes code maintenance much easier ([21]).

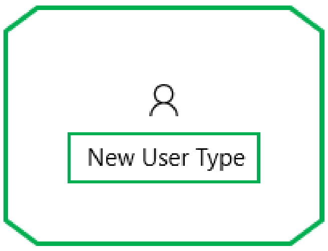


Figure 3.5.: View of an user type

The user type view follows the design of the view of an object type, as mentioned Figure 3.5 is used to illustrate this. In comparison to the object type the user has a border that is colored green and a little pictograph that shows a human being so that it can be distinguished faster.

Similar to the object type, the user type demands the setting of some special attributes too. In this case the attributes to set are a “password” attribute and a “name” attribute. Analogous to the object type view the setting also takes place in the split view. This is depicted in Figure 3.6.

Another important point is that a user type can have roles added to itself. This roles are shown after selecting the vertex. Because this is a core feature, it is depicted in an own section. Refer to Section 5 for detailed information.

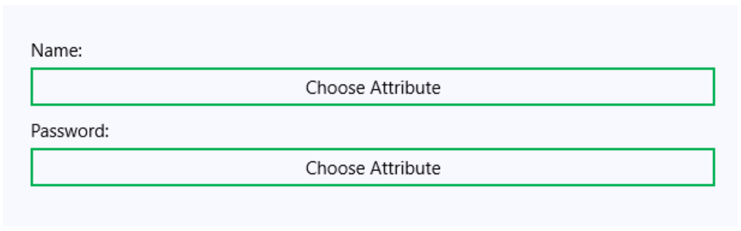


Figure 3.6.: Special attributes for a user type

Selection

Both the user type as well as the object type can be selected. The selection can be done natively by clicking on the object and it will change the color of the object border to gold. The difference between a selected and a not selected object type can be seen in Figure 3.7. This dying is used to give the user visual feedback and the possibility to check if he has selected the right element. For both elements the selection will trigger the opening of the split view as previously mentioned. This enables the user to insert additional information for the selected element or to delete it. The deletion of an element has to be confirmed via a dialog to prevent an unintentional deletion.

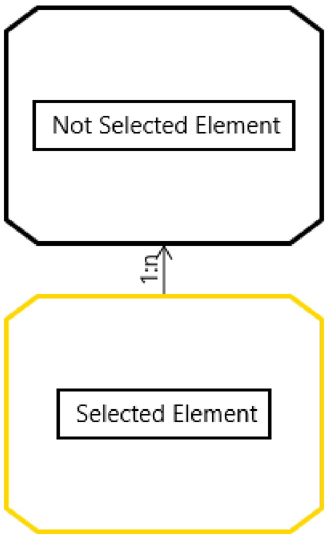


Figure 3.7.: Selecting of an object type

Attributes

Every object type and user type can have different attributes, which are used to automatically generate a form at run-time. Also they are used in the micro process to model the behavior of objects in different states and how different attribute values can change the course of the process. The application supports different kinds of attributes, those are depicted in Table 3.1.

ATTRIBUTE TYPE	EXPLANATION
string	usable as input field for free text or combo-box with defined values
boolean	usable in form of a checkbox
number	usable as input field to insert a number that can be chosen freely by the user, supports the determining of an acceptance range or only a single upper/lower bound, supports decimal or integer input, the data is stored as decimal on the server in any case
date	usable to generate a date picker
reference	usable so that the user is able to set a reference to one specific instance of the object type or user type that is defined in this attribute
relation	relation attributes can only be named and are created by the server according to the direction of the relations the user modeled between different object types and/or user types

Table 3.1.: Attribute overview

Besides its own specification, each attribute requires a name by which it can be identified and used in other parts of the application. There is no limit on how many attributes can be created by the user. They also can be deleted individually at any time and their name and specs can be changed at will.

How attributes are displayed for the user is shown in Figure 3.8. The figure shows how the pane of the split view that opens after an object type or user type is selected looks like. With the button in the upper right corner, additional attributes can be added by the user. Each attribute is displayed in its own area that is modeled as a window. This allows the user to view them all together but also do not get confused because the attributes are visually distinct. Every attribute can be deleted manually by clicking on the red button in the right corner of the attribute window.

The attribute concept is open for extensions in the future. Every extension only need its own view which enables the input of the specific option and a counterpart on the server. In the future it might be possible to allow not only the pick of a specific date but also the pick of an exact point in time. So that time critical processes can be supported as well. Also could be possible to implement a data-upload attribute that gives the run-time-user the option to upload process relevant data like pictures, documents or similar files. After a successful upload the data could be attached to the process itself.

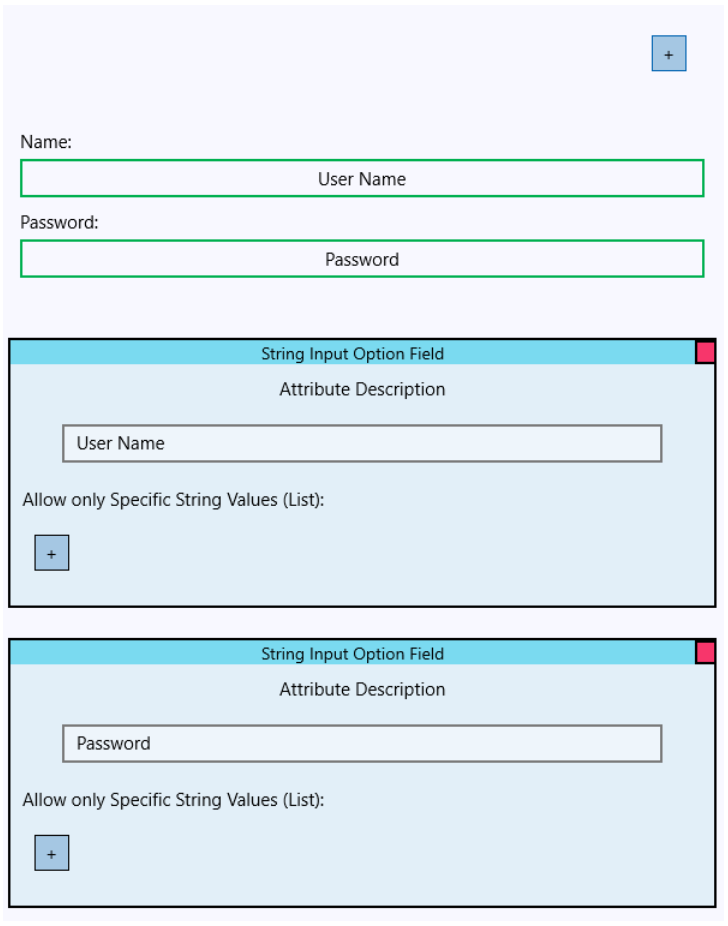
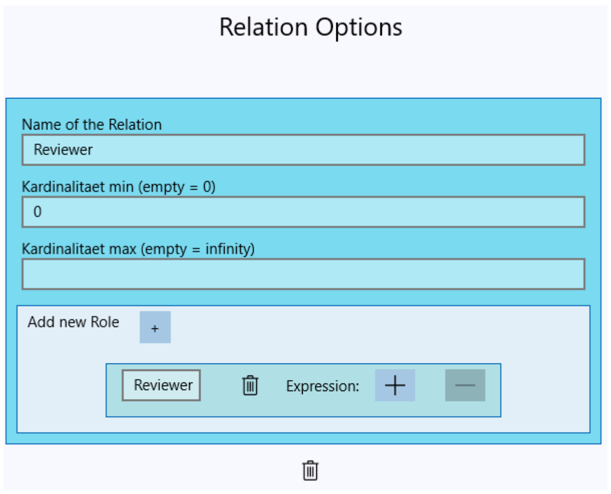

+
Name:
User Name
Password:
Password
String Input Option Field
Attribute Description
User Name
Allow only Specific String Values (List):
+
String Input Option Field
Attribute Description
Password
Allow only Specific String Values (List):
+

Figure 3.8.: Input option for the attributes of an object type

3.2.2. Edge

Within the data model there is only one kind of edge that edge is called “relation type”. A relation type is used to model the interdependence between different object and/or user types. The relation type is generated automatically by clicking on a vertex and dragging the pointer a short distance. It is dropped automatically if the user does not hit any other vertex. However, if the modeler drags and hits another vertex the relation type is connected to the nearest anchor of that vertex and both are connected.

Similar to the both object type and user type, a relation type can be selected as well. The relation has different attributes that are needed. The first attribute is the name of the relation, the second and third attribute are the minimum cardinality and the maximum cardinality. The cardinality is there, because it is planned that the application can support many-to-many dependencies that are forbidden at the moment. To prevent adaptation in the future, both values can be set at the present time even if they are not used by the system yet. If the edge has a user type as a target it also offers the option to add roles to the relation. For further information about roles, look at Section 5. Figure 3.9 shows how the options are displayed to the user.


Relation Options

Name of the Relation: Reviewer

Kardinalitaet min (empty = 0): 0

Kardinalitaet max (empty = infinity):

Figure 3.9.: Input option for a relation between object type and user type

3.3. Micro Process Modeling

The micro process model allows the user to model the course of the process in detail. Therefore, the modeler can specify in which order values have to be inserted and which permissions are needed in order to fully execute the process.

3.3.1. Vertices

Within the micro process model three kinds of vertices are needed. All vertices (cf. Appendix B, Fig. B.2 (1-3)) can be dragged from the sidebar.

State

The micro process state (cf. Appendix B, Fig. B.2 (1)) is the first type of vertex which can be placed on the grid by the user. It is also special because it serves as a new placement grid as well, because within each state micro steps can be placed. As it was pointed out in the overview, a state is used to depict a specific point in time for certain object or user types. The information a state view model provides include the id, a state name, the micro steps that are placed within the state and incoming as well as outgoing backwards transitions. Of those attributes, only the name can be edited by the user, because the other options will be set by the server and can only be influenced passively by the user as they depend on the other parts of the model. Also every state contains information about the permissions that are required for it, refer to Section 5 for more detailed information. An empty state is depicted in Figure 3.10.

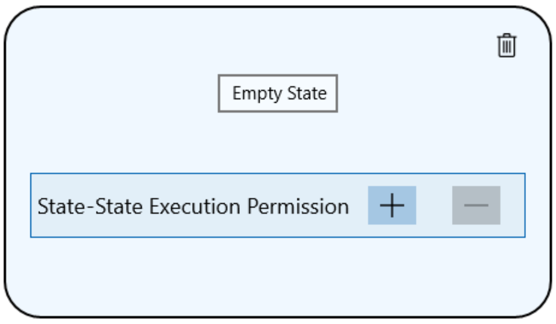


Figure 3.10.: Empty state in a micro process

As it is shown in the figure each step can be deleted by clicking on the button in the upper right corner of it. The deletion will be done after a user dialogue that asks for confirmation from the user before finally deleting the state. Besides the confirmation it also is necessary that the state is empty.

Micro Step

The second type of vertex is the micro step (cf. Appendix B, Fig. B.2 (2)). A micro step can only be placed within a micro state and it represents an attribute of an object or user type from the data model the micro process belongs to. Figure 3.11 illustrates the view of a micro step.

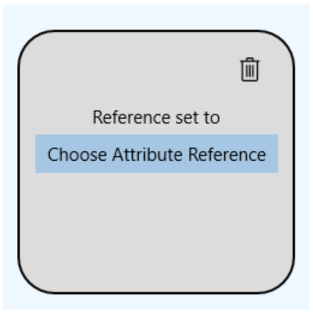


Figure 3.11.: Micro step placed in a state

The first button allows the user to set the attribute reference. Because an attribute can only referenced once per state it is mandatory to check which attributes are not referenced and therefore not in use. This is done automatically by the system and only available attributes are displayed for the user to choose.

A value step can be dragged into the micro step. As it is in the design philosophy the button to delete the micro step is also in the upper right corner and the deletion can only be done if there is no value step placed within the micro step.

Value Step

The last vertex is the value step (cf. Appendix B, Fig. B.2 (3)), like the other vertexes it can be dragged from the sidebar and will then be placed automatically by the system if the user drops it to a micro step. Because dropping something on a wrong position is prevented by the system, the modeling flow will not be disturbed and wrong user-actions are prevented automatically. As it is shown in Figure 3.12 each value step can either be empty or have an expression. This expression allows the user to specify different possibilities within a micro step. Besides the deletion button, that has no restriction besides the user safety check because the expression is not a nested object, but an integral part of the value step and therefore has not to be deleted first. The value step contains a button to save the value of the expression to the server. This is necessary because while modeling the expression itself, it will be changed constantly. If every of those changes is sent to the server this will generate unnecessary traffic that can be avoided with the save button.

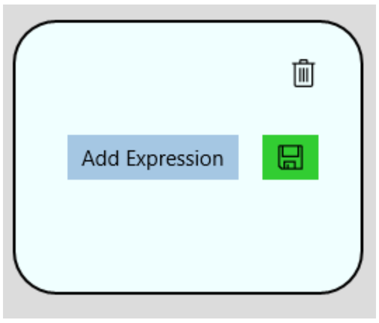


Figure 3.12.: Value step with expression

Future Extensions

At present time the modeling tool is built and optimized for the usage with a mouse and a keyboard, but this can be changed in the future. One of the advantages of the windows universal platform is the fact that an application does also work on mobile devices like tablets or mobile phones. While a phone display might be too small to model on it a tablet is sufficient. Therefore, the drag and drop might not be the best option. In future adaptations of the application it is recommended to use an on-pointer-pressed menu like it is known from general applications that run on mobile devices with “Windows 10”. Due to the design this adaptation is very easy and only the insertion methods have to be changed.

3.3.2. Edges

Within the micro process two different types of edges are supported. Both types serve different purposes and are illustrated in a different way.

Micro Transition

A micro transition is used to model the normal execution path of a process. It is depicted as an arrow with a solid line and can be dragged from a value step to a micro step or from a micro step to another micro step. All other combinations, like micro step to value step, micro step to state or value step to state are not allowed and therefore are prevented by the system automatically. Micro transitions have different options that are shown in Figure 3.13.

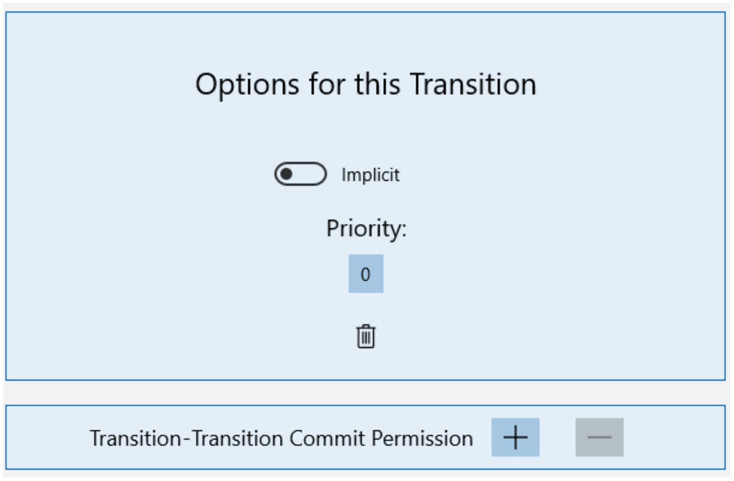

Options for this Transition
Implicit
Priority:
0
Transition-Transition Commit Permission
+
—

Figure 3.13.: Micro transition options

First of all, every micro transition can be implicit or explicit, which can be changed by the user through a switch. If a transition is implicit this means that the process will move to the next state automatically as soon as the implicit transition becomes activated. If it is explicit it means that the movement to next state cannot be done automatically but has to be confirmed by the user. Also a priority can be chosen for all transitions that have the same micro step as source point. With that the system can determine which transition has to be executed if there are multiply possibilities because only the highest priority will be chosen.

Besides that, general information, including the permission to commit, can be specified as it is depicted in Figure 3.13. This permission allows the holder to commit the generated form and move one state forward in the process.

Backward Transition

The backward transition specifies to which state the process can move back if the user decides to revert an already executed macro step. For example, if there is a backward transition between the state “Occupied” and “Published” but not between “Occupied” and “Closed” the process can only move to the state “Published” if it is reverted in the state “Occupied”. Each backward transition is depicted as an arrow with a line of slashes. It can only be drawn from state to state and the source step has to be in the set of successors of the target step.

After selecting a backward transition, the user has the option to edit the backward jump permission, this is depicted in Figure 3.14. Without this permission a user cannot move back.

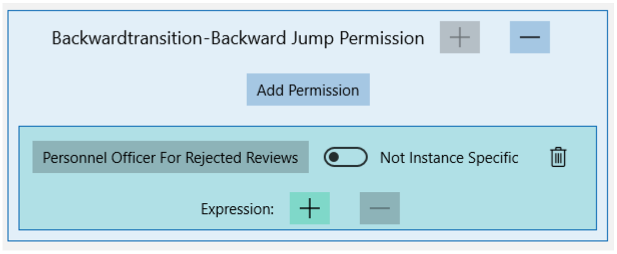

Backwardtransition-Backward Jump Permission
Add Permission
Personnel Officer For Rejected Reviews
Not Instance Specific
Expression:

Figure 3.14.: Backward transition options

3.3.3. Placement on the Canvas

The canvas placement of the micro process model is a bit different from the other two models. Because as mentioned a state can contain micro steps, this means that there is one placement algorithm that places states and external relations on the grid of the graph and a second placement algorithm that places micro steps within a state, as well as internal relations between steps. The placement of the value steps is not done by any algorithm because they are just put in a stack panel and therefore are placed in a vertical row within the micro step. This is possible, because there are no relations between value steps. A newly generated micro state or micro step will always be placed in the lower-left corner of the grid (cf. Appendix B, Fig. B.1 (5)) because this is the most comprehensible place to that a user would look after a new element. On another position the user would have to search for the element and his creative flow would be disturbed. Figure 3.15 shows a complete set of vertices, even a newly generated one and gives a brief overview over the single elements and how they are placed. As it is shown there the empty not connected step is in the lower left corner while the empty step that is already connected to an existing step is placed on the same level as the step to whom it is connected and that already contains information.

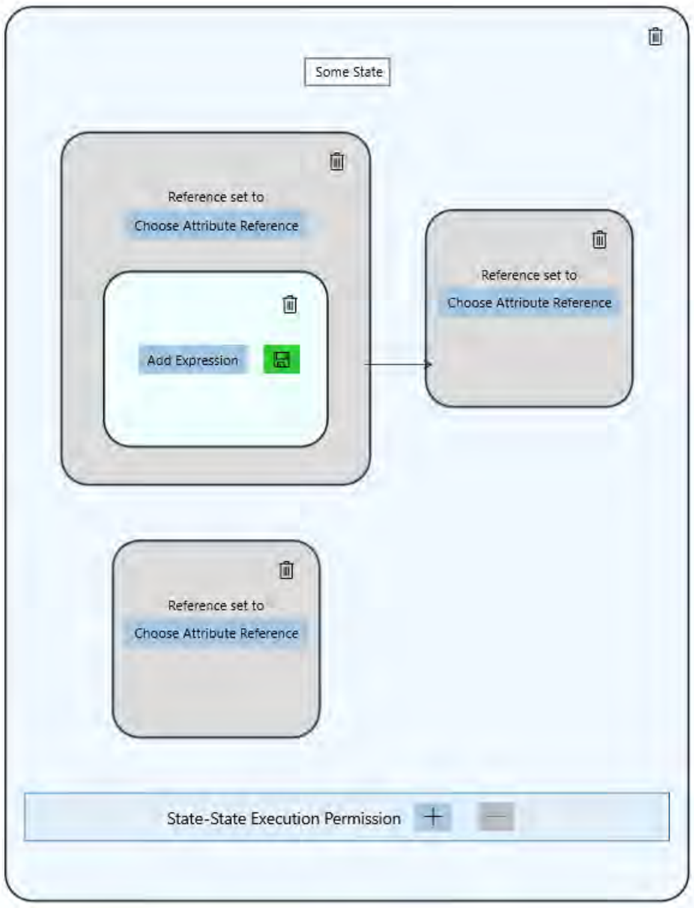

Some State
Reference set to
Choose Attribute Reference
Add Expression
Reference set to
Choose Attribute Reference
Reference set to
Choose Attribute Reference
State-State Execution Permission

Figure 3.15.: Micro process vertices on the canvas

3.4. Macro Process Modeling

The macro process specifies the interdependence of objects defined in the data model. While its significance for the semantics of the model is very high the vertices and edges that have to be modeled are very simple.

3.4.1. Macro Step and Ports

The vertex in the macro process model is the macro step (cf. Appendix B, Fig. B.3 (1)). It differs a bit from other vertexes because it has two parts. As it is depicted in Figure 3.16, each macro step has at least one port (little white rectangle with rounded edges, at the left side) attached to it, it is important to note that start and end steps have a special marking. For the start step this marking is the color green and for the end step the color red. The port is used so that the user is able to model either an “AND” or an “OR” semantic, as it is explained in Section 2.1. It serves as target for a transition that is drawn to the parent macro step. After the transition is connected, the macro step automatically generates a new empty port that will be shown below the old port. This process can be repeated indefinitely.

The macro process has a reference to a specific state of a particular object type from the data model. For that reason, the user can first pick the object or user type and after selection one of those the button to choose a state will be activated. It is important that macro steps can not be connected if they haven’t got the state set.

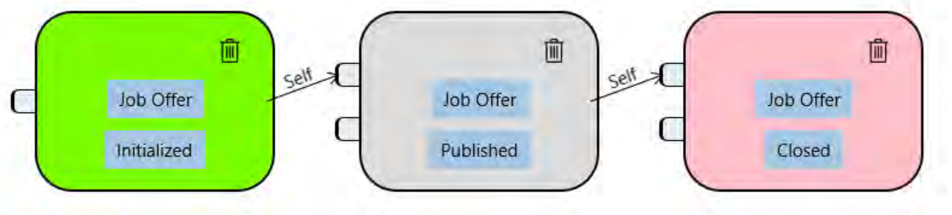


Figure 3.16.: Start, intermediate and end macro steps with ports

3.4.2. Macro Transition

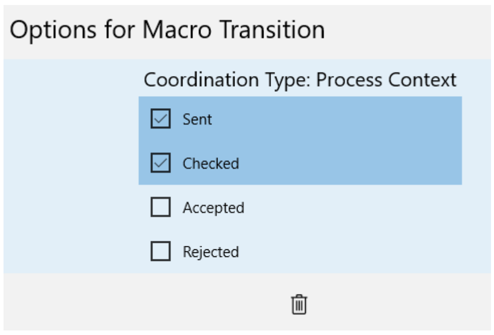
Besides the typical usage of an edge to model how vertices are connected to one another the macro transition also contains information about the coordination type. This type is used to specify relations on a greater scale within the whole data model. There are four different types that are determined automatically by the system based on the graph that the user created of the data model. All types are depicted on the edge.

Each edge can be selected and deleted in the pane of a split view that will be shown after the selection. The brush of the edge is black, after selection this is changed to gold. Each macro transition can only have a macro step as source and a port as target. An edge can only be drawn once between two macro steps. If an edge already exists nothing will happen. Furthermore, illegal targets are greyed out and can not be connected.

Process Context - Top Down

With the process context coordination type, the user can model, how the lower level instances of a data-model depend on the state of a higher level instance. If, for example, an application depends on a job offer, it can be created while the job offer is in the state “initiated”. After the job offer leaves this state normally it is not possible to create new applications. This behavior might be not correct, for this reason there is the possibility to allow the creation in specific other states as well, for example in the state “published” ([11]).

This choice is depicted in the form of a list view, as it is illustrated in Figure 3.17. This way, the user can select the states easily and very fast without having to scroll through the graph and select states manually or try to remember their name. It is important that, due to the data structure that the application receives from the server, the initial state has to be always selected. In case of the example this would be the state “initiated”. Because this is mandatory the value is automatically set without any way for the user to influence that.


Options for Macro Transition

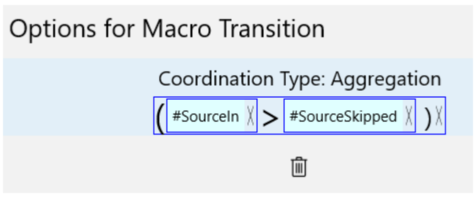
Coordination Type: Process Context

- ☑ Sent
- ☑ Checked
- ☐ Accepted
- ☐ Rejected

Figure 3.17.: Coordination Component - Process Context

Aggregation - Bottom Up

The coordination type “aggregation” specifies how lower-level objects depend on higher-level objects. To allow the user a detailed specification, it offers the uses of an expression. For further reference on expressions, refer to Section 4. A value for the expression is set automatically by the system when it determines which type of coordination component the edge is. But it can be edited by the user at any point in time, the changes will be saved as well. Figure 3.18 shows how the information is presented for the user.


Options for Macro Transition

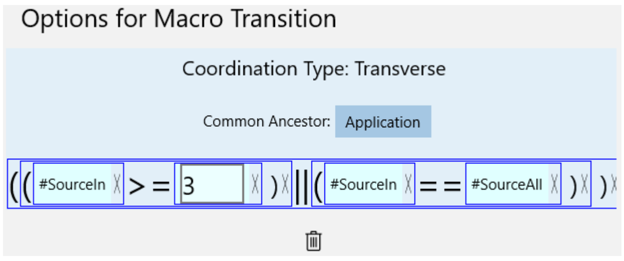
Coordination Type: Aggregation

(#SourceIn > #SourceSkipped)

Figure 3.18.: Coordination Component - Aggregation

Transverse

Akin to the coordination type “aggregation” the coordination type “transverse” comes with the possibility to use an expression. Besides this expression the user can also determine a common ancestor object by the use of a button as can be seen in Figure 3.19. This object is needed because the “transverse” component describe how objects are indirectly dependent on one another. For example, the object “review” depends on an application, as well as a job offer. If the “application” is the common ancestor, the semantic would be that if there exist a specific number of positive reviews within the whole application the interviews can be done. Otherwise, if the common ancestor is the “job offer” the semantic would be changed to the fact that interviews can only be held if there is a specific number of positive reviews for one specific job offer. Because the choice influences the semantic dramatically it has to be decided by the user. The click of the button will show the modeler the different choices that exist. This method prevents the user from having to change the graph to find the correct object because only those objects that are possible will be presented.


Options for Macro Transition

Coordination Type: Transverse

Common Ancestor: Application

((#SourceIn >= 3) || (#SourceIn == #SourceAll))

Figure 3.19.: Coordination Component - Transverse

Self

The component “self” is the simplest component. There is no need for the user to input any variables because this type has none. It is only depicted for informational reasons and the user only has the option to change the self type to a self-transverse type if the macro state does not refer to the primary object type. For example, if the macro process is part of the object type “job offer” all macro transition types that are marked as self and refer the “job offer” object type cannot be changed while others can because they refer another object type. Figure 3.20 shows how the self type is depicted for the user.

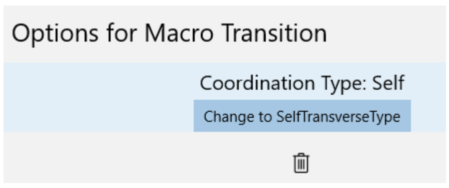


Figure 3.20.: Coordination Component - Self

Self-Transverse

Self-Transverse is a special variant of a transverse type that can only applied to a self type by the choice of the user. In this special case the common ancestor is clear because it is the object type of the self type. Besides this the user can edit an expression, like he can for the normal transverse type. The self type is modeled if an object depends on itself. An example for this can be seen in an application process. If there is only one job offer but many applications, there can only be one application accepted. As soon as one is accepted, the system has to be able to prevent the acceptance of further applications this can be modeled with the self -transverse dependency of the object type. Figure 3.21 shows how the interface for a self-transverse type is presented for the modeler. As can be seen, the user has also the option to change the self-transverse type back to a normal self type by clicking on the change button.

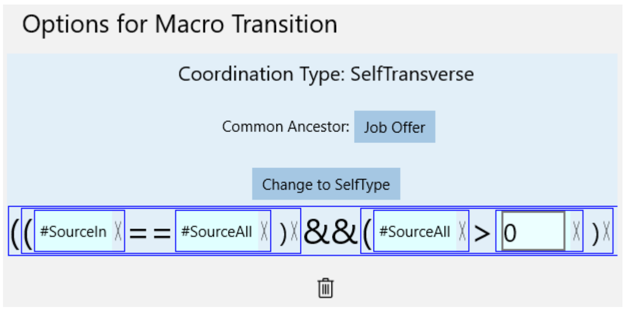

Options for Macro Transition
Coordination Type: SelfTransverse
Common Ancestor:
Job Offer
Change to SelfType
((#SourceIn == #SourceAll) && (#SourceAll > 0)

Figure 3.21.: Coordination Component - Self-Transverse

4 Expressions

Expressions are a very important part of the application. They are used in every graph and they have to follow the same design because only then the user has a high recognition factor while working with the application. Because of that, they are one key feature, as mentioned in Section 1.3. This section shall give a brief overview on how and where expressions are used within the application (Section 4.1) followed by the design and implementation on server and client (Section 4.2 and Section 4.3).

4.1. Use within the Application

As was said in the introduction, expressions are used widely within the application. They are listed in more detail in this subsection.

- Data Model
  - Context Roles
  - Relation Roles
- Micro Process
  - Value Steps
  - Permissions
- Macro Process
  - Coordination Components

It is important to remark, that besides the parts of an expression that are used to model, the coordination components and all the variables, the concept can be used without the application itself because it is designed generically and only the features that are necessary for the PHILharmonicFlows concept are a non-generic implementation. In theory they could also be used without the application though. But they would lose their contextual meaning and therefore be irrelevant, however they are not bound to the data that is used by the application and the server.

4.2. Concept - Server side data structure

In the basic concept of PHILharmonicFlows ([11]) expressions were used in the value step types. There it was suggested that a user can use different expressions, like comparative operators, system variables, system functions and logical operators to specify the options of a value step. While the idea is good and it gives many possibilities to model a complex process, it is not very intuitive for the user to insert an expression that is text based. More importantly a text based expression cannot be evaluated automatically because of different reasons, that are explained in more detail here.

- All objects are referred to by a generated ID - and there is no possible way to match a given text input like a name to the right ID also it is impractical that the user inserts the ID instead of the name because such an ID is composed of ten single digits. No user would be able to remember or even compare them easily.
- Some operators like “less than or equal to” can be written in different styles like <=, $\leq$ or =< . While the first two options are common, an inexperienced user could try to write the third option. This will cause some major problems, because either the expression cannot be evaluated at all and throws an error, or the programmer has to think about every possible way a user could insert such a simple operator, which would increase expenditure for development, as well as maintenance. This would be uneconomical and therefore has to be avoided.
- Not every variable or constant that can be used supports all operators. For example, there is no appropriate use of string-values in combination with a boolean-operator like “OR”, so besides the fact that the validity of the expression can not be determined by the system it cannot prevent errors in advance.
- The return type of the expression is based on the used operator and it is possible to use an expression within an expression. With a text-based expression it costs way too much effort to for the user to model the expression correctly. Moreover, it would be very hard to determine the value by reviewing the expression manually, because there is no graphical support that makes the different expression parts stand out.

To address all of the problems mentioned above, the modeling tool uses a modeler for the expression which gives the user the freedom to choose every possible operator, variable or constant while securing an automatic validity test because all user inputs are saved within a generated expression-class-framework.

This framework is designed to support expression-functions with any arity the user wishes, as long as he implements the necessary structure first. Because PHILharmonicFlows is designed to use only nullary and binary functions at the moment the arity support is limited to a maximum arity of three. This means that the maximum function that can be used is a ternary. But it is possible to extend the framework. For that, the user has to add the necessary classes so that n-ary functions are supported by the server side data structure. The view for n-ary functions is already implemented and fully functional.

An overview of the complete serverside-framework can be seen in Figure 4.1.

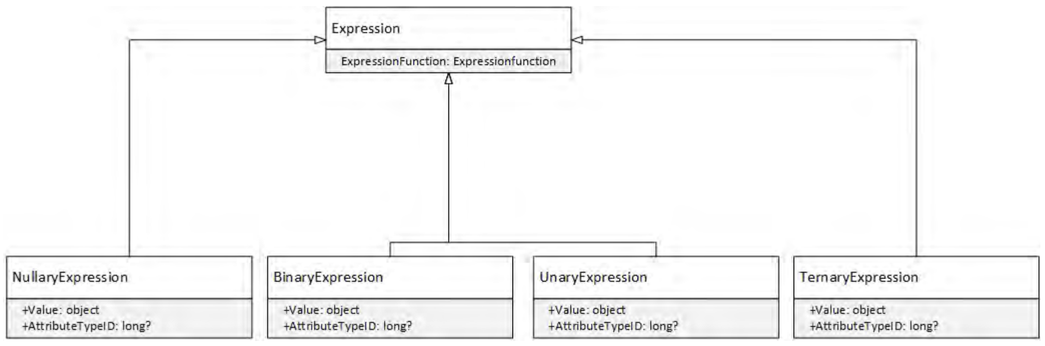


Figure 4.1.: Overview of the expression framework (server side)

As it is depicted in the overview there is one super-class, called *Expression*, that has different sub-classes from *NullaryExpression* to *TernaryExpression*. All further possible n-ary functions have to be implemented as classes at this point. The class *Nullary-Expression* is the super-class of the variables, constants and coordination-expression-parameters that are used within PHILHarmonicFlows and as such is very important. Constant and variable nullarys support the following datatypes.

- Boolean
- Date
- Number
- String

For the coordination nullary functions the following parameters are implemented.

- SourceAfter
- SourceAll
- SourceIn
- SourceBefore
- SourceSkipped
- TargetAfter
- TargetAll
- TargetIn
- TargetBefore

- TargetSkipped
- TargetAfterGlobal
- TargetAllGlobal
- TargetInGlobal
- TargetBeforeGlobal
- TargetSkippedGlobal

For a detailed reference about their use and functionality within the coordination components, look at Section 3.3.2.

Because every arity needs its own expression-functions, these are specified in an enumeration and have an internal value-representation for every member of the enumeration. Furthermore, they come with a string representation to depict them in a readable way for humans and a number that specifies which arity is supported.

The binary functions supported by the system are listed in Table 4.1.

FUNCTION - STRING REPRESENTATION	FUNCTION - SYMBOL
less	<
less or equal	<=
greater	>
greater or equal	>=
equals	==
not equals	!=
or	||
and	&&
xor	^
implicates	->
plus	+
minus	-
multiply	*
divide	/
modulo	%

Table 4.1.: Binary functions overview

Because normally unary and ternary functions are not needed there are only two implementations yet. Those are used to show and prove that the concept works in showcases. For the unary this is the "not"-function (!(x)) that negates a boolean value and for the tenary the example is f(x,y,z) which can represent for example a function to calculate the volume of a cube.

4.3. Concept - UI-Side

In order to allow the user to understand and model an expression, a UI concept was created.This framework is illustrated in Figure 4.2. It is important to note, that this framework satisfies the formulated requirement “Simple Expression Modeling” (REQ-8).

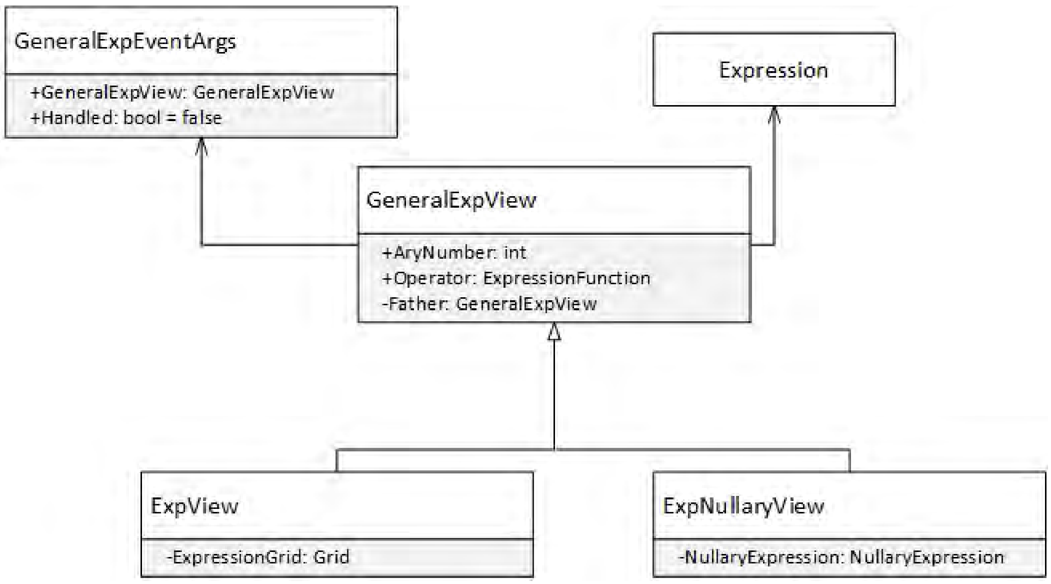


Figure 4.2.: Overview of the expression framework (UI-side)

As it can be seen in this figure there is an abstract super-class (*GeneralExpView*) that contains the general information that an n-ary function need, to be built and illustrated correctly. This information includes the arity-number, the enumeration-value of expression-operator as well as the string representation of the operator and a protected field for the parent object of the expression. This last field is needed for the expression nesting. Also the class defines general functionality that the view of an arity has to support. This includes the option to extract the data structure of the expression from the view and a reset function in case the user decides to delete the expression or parts of it. Furthermore, to allow the deleting of parts of the expression, it defines an *EventHandler* that uses a userdefined eventarg as option. The last part of the class is a support function which is used to display a list of strings and allows the user to select one of those. This function is used heavily to allow user-input for the expression.

Within the overview there is also a customized class that extends the class *EventArgs*. This class is used so parts of the expression that are nested within each others can be deleted safely. The class contains a property for a *GeneralExpView,* which saves a reference to the instance that invokes the event. Therefore, it places the class reference and a boolean property. This property is used to signalize if the event is handled or not. For example, if three classes are nested within each other and the last class invokes the event, both the second and the first will try to handle the event - because they have event handler set to those particular events. This would lead to a deletion of both the second and the third class but this is not what the user intends to do. To prevent a complete deletion, all classes check if the boolean property still has the value “false”. If this is the case, action will be taken and the first class that handles the event, in the example the second class, will set the property value as true after completing its work. All other classes will not handle the event anymore, because the value for the “handled”-boolen is now “true” and the deletion works as it was intended. The implementation of this concept is shown in Listing A.1.

4.3.1. Class - ExpView

The abstract view is implemented by two classes that are used to illustrate the construction of the expression-view for the user. The first one is the *ExpView*. Within this class the skeleton of the expression is depicted. This includes the possible nested hierarchy of different forms of arrities as well as a control structure to delete parts of the expression and the possibility to evaluate the depicted expression from the view and generate a data-structure-object.

The class can be build empty, then it allows the user to model the expression from the scratch. Otherwise, it can have an expression as input parameter. In this case it will illustrate the given expression by building it. After this, the user is allowed to adapt or delete it as it is possible for any self-modeled expression as well. The actual viewable part of the class is constructed fairly simply, as can be seen in Listing A.2.

It basically only contains a *Grid* that can be used to display the different stages of the arity choosing and a button that can be used to delete the expression. A deletion will reset the view to the initial form which can be seen in figure 4.3.

Figure 4.3.: Initial form of the ExpView

After the user clicks on the view, all available arrities will be shown, as it is depicted in Figure 4.4. As the next step the modeler can choose one of them. The choosing process is done by a message window because this allows the user to clearly select one specific arity, also it can be easily extended if there will be more available arities in the future. The choice brings the view to a new stage. In that stage the text “Choose Arity” is changed to “Choose Operand” and, therefore, the user can choose an operand for the expression. After the operand is determined, the view changes to the form that is depicted in Figure 4.5 and Figure 4.6 and the user can specify the expression in detail.

It is important to note, that a choice cannot be changed after it is done. The only way to change it is to reset the whole expression or the part in which the arity function has

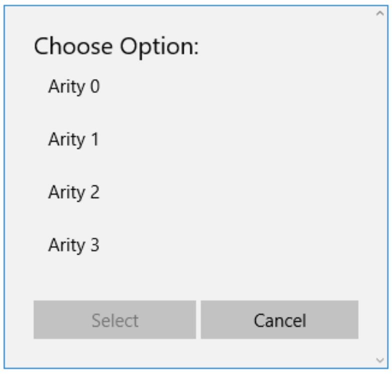


Figure 4.4.: Selection of an arity

to be changed. This is due to the fact that there are two styles in which a view can be depicted that depend on the arity. Normally the expression will be depicted in the form that the operator is shown, followed by the list of possible operands that can be seen in Figure 4.5 for a ternary.

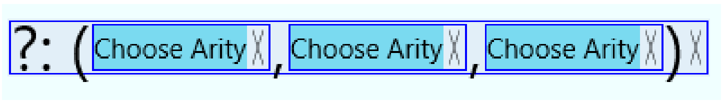


Figure 4.5.: Empty view of a ternary

However, in case of a binary function this is not the form that the modeler would expect. Because it is not natural to write a binary function this way and so a user cannot understand it immediately. For that reason, a binary function is depicted in the form Operand-Operator-Operand. This version enables the user to understand it faster and it is also more intuitive. Figure 4.6 shows how a binary function is depicted for the user.

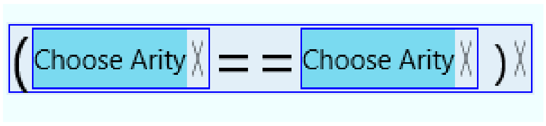


Figure 4.6.: Empty view of a binary

4.3.2. Class - ExpNullaryView

The class *ExpNullaryView* is a special form of the class *ExpView* because it is structured in the same way but it is ensured that there are no more sub-classes nested within because the nullary function is defined as a single value. This value can be fixed or variable but it is only a value and has no further operator structure. In Section 4.2 the different possible options for nullary functions are explained. For each of those nullary functions, a specialized view for the user to insert the value is needed.

Coordination Expressions

For the coordination expressions no special user input is needed, because the expression works without any input-value. The value is given trough the context in which the expression is used, for more detailed information confer Section 3.4. This is the reason why at this sort of nullary only a text is depicted that shows which nullary-function was chosen. Figure 4.7 shows the view with the coordination “source all” function as an example.

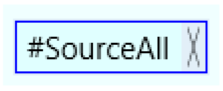


Figure 4.7.: Example coordination nullary - Skipped

Variables

Nullary expression variables demand the user to set the reference to a particular attribute that will be used as value base for the nullary function. Therefore, a nullary variable has a button which allows the user to choose from different available attributes and will display the name of a chosen attribute as its content. This is illustrated in Figure 4.8.

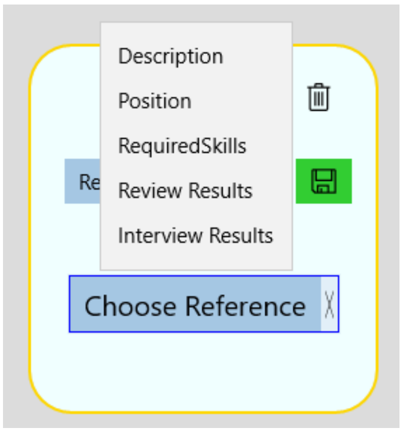


Figure 4.8.: Example variable nullary

All references that are shown as possible targets are supplied by a list of attribute view models and the selection process is completely variable. All options are shown to the user in a fly-out menu that is generated on the fly, the code can be seen in Listing A.3. After selecting one available option, the name is not only depicted as content of the button but the id of the chosen attribute is also saved in a nullary-expression-variable object that is returned if the expression is evaluated.

Because attributes only exist within the context of a specific data model, it is possible for the user to try to model an expression without a variable. The modeler is not limited and the user can choose from all supported options even if they are not available at the moment. This would lead to an exception if the user tries to refer an attribute that does not exist because it is not created at the moment or it is deleted after the reference was once set. To prevent the application from crashing if no attributes exist and also to signal the user, that a reference is incorrect, the view of a variable verifies if a correct reference exists within the application because it cross-checks with all available attributes. If there is no match or no attribute at all the view will reset and signal this fact for the user. This is illustrated in Figure 4.9.

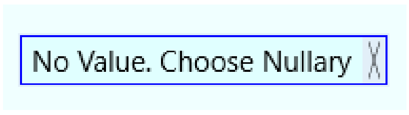


Figure 4.9.: Example variable nullary with no available or incorrect attribute references

Constants

In contrast to the nullarys that are used for the coordination components and the attribute variables, constants are more complex, because for every data type that can be inserted there has to be a different input method. Due to this fact there are different implementation to insert constant-values.

Boolean Constant

A boolean constant can have one of the two values “true” or “false” because of that the chosen input option for the user is a checkbox. With that option he can only click and not set any wrong value like 0 for true, 1 for false or have any typos. Also the value need not be parsed, but can be read and written directly. Figure 4.10 shows the input option for a boolean constant.

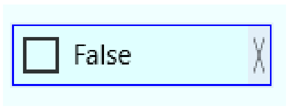
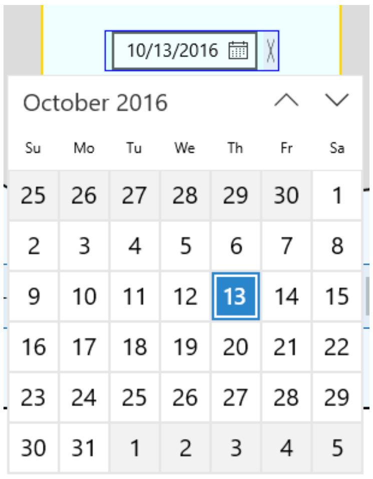


Figure 4.10.: Example nullary for a boolean constant

Date Constant

By using the date constant, the modeler can choose one specific date to use it in the expression. Within the UWP framework there are different sorts of time controls. First, there is the “calendar view” that allows the user to pick a single date or a range of dates, second the “calendar date picker” that allows users to pick a single date from a contextual calendar, third the “date picker” that is used to pick a single date without any contextual info and last the “time picker” to specify a single time value [23]. The application does not need the user to be able to pick a range of dates or specify a time. Furthermore, the context information is a valuable support when choosing a date due to the fact that the day of the weak is visible. Because of that the “calendar date picker” is the option that was chosen for the date constant. This option is also the most compact which is better because it helps to make the expression more compact and therefore supports the comprehensibility.

Figure 4.11 shows how the date constant is presented for the user. The upper half is the usual form that is displayed, after a click on the icon the calendar expands and the user can change the month or select a date that is shown.



25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Figure 4.11.: Example nullary date constant - with expanded calendar

String Constant

The string constant allows the user to enter any text he likes. This text will be taken as input. The input variant for this type of constant is a text-box. There the user can enter as many characters as are needed, the text-box size will be automatically adapted to the user input and there is an option that allows the user to clear the box. Figure 4.12 shows the input for a string constant.

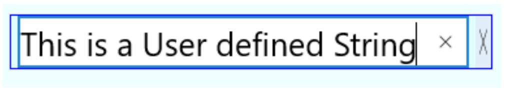


Figure 4.12.: Example nullary string constant

Number Constant

Number constants are inserted with the use of a text-box as well and they are basically depicted the same way as a string constant. But there are two important differences. The first one is that the value of the expression has to be parsed from double to string if the expression modeler generates the view from an existing expression-number-constant-value. Furthermore, it has to be parsed from string to double if the user enters a new value in the text-box and tries to save the value as an expression. Because the parsing is a possible source of exceptions if the user is not careful the second difference between the number and the string constant is, that the text-box for the number constant will only allow very specific input. As it is shown in Listing A.4 the text-box has its text changing event specified.

This event is called automatically by the text-box if the user tries to enter text and the actual text is in the process of changing. To prevent wrong input like characters or ensure that only the English version of the decimal separator can be used the input is validated by a regular expression. This expression ensures that there are only numbers followed by one dot followed by numbers inserted in the text-box. With this check parsing the value is always safe because wrong input cannot be entered and the parsing takes place in the text changed event. This event is called after the text changing event.

Because the text-boxes are expanded automatically by the application, it could be that the box enlarges not only itself but also the elements that contain it. This can lead to the problem that the edges of a graph are not at the anchors anymore because the anchors are moving with the other elements while the edges do not follow automatically. To prevent this each text-box supports an event that is called when the box has lost its focus so that edges can be rerouted in case that the box was enlarged.

4.3.3. Future Improvements

In the future different adaptations and improvements can be made for the expression modeler. The first one is that it should be possible for the user to choose the way an expression is displayed. While the inline depiction is optimal to save space, in other cases a tree-like view could be better to understand for the user and could be used if the vertical space is not limited.

Furthermore, in the future, it should be possible to save structural information about how an expression has to be depicted. For example while for a function like f(x,y,z) the version that is used at the moment (operator followed by a list of operands) is satisfying for other ternary operators like if-then-else it would be better to display the expression in the form “if operand then operand else operand” because this is the way a human normally imagines the operation.

The last suggested improvement is to make the modeler more variable so that the user can drag and drop parts of the expression to another place. This might be a good adaptation because it allows to model more freely.

5 Roles and Permissions

“Process and data authorization is based on user roles” ([11]) and each of those roles can have different permissions, which can be specified. This section shall give an overview on how the system for roles and permissions, the second key feature, work, within PHILharmonicFlows, the old concept which was introduced with the original idea of PHILharmonicFlows is illustrated in Section 5.1. The following Section 5.2 shows the new version of this concept, how it is implemented in the application and how the above specified requirements (REQ-9, REQ-10) are satisfied.

5.1. Old Concept

The concept pointed out how important roles and permissions are in a Business Process Management System. Figure 5.1 illustrates the idea, it shows the authorization table for the object type “review” as dark-gray box. Within the object type all state types are depicted as light-gray boxes and within each of those states the attributes and relations of the state’s object type are illustrated in a white box. On the top of the object type box there are some attachments in an even darker gray, those symbolize the different roles that exists within the data model. In the columns below the roles there are the different permissions listed per role.

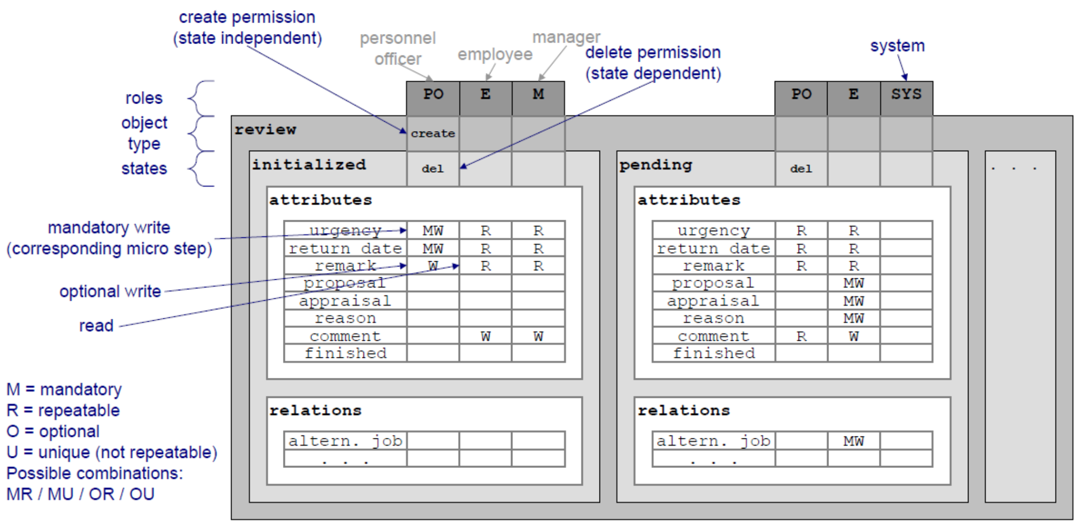


Figure 5.1.: Authorization table ([11])

5.2. New Concept for Roles and Permissions

While the idea from the original concept allows the specification of all possible roles and permissions and shows them in a single table this is also the greatest disadvantage. With this method the user loses the overview and might not be able to do a correct modeling of all roles and permissions that are necessary. Therefore, the concept was reworked to allow the modeler to handle roles and permissions in a more natural way.

5.2.1. Roles

All roles can be specified in the Data Model, within the modeling process the user has the option to add roles to a user type, as well as to a relation between an object type and a user type. Each role contains a role-name, a list of permissions that will be explained in detail in Section 5.2.2 and a role-context-expression, which is optional. With this expression the role can be specified in detail because contextual information can be modeled. For example, it is possible to specify, that in an application process, a user with the role “HR Germany” only has access to job offers in the region of Germany, but not for other possible regions ([11]). Each specific role contains additional information to which user type or relation it belongs to.

Relation Role

A relation that has an object type as source and a user-type as target can have a relation role added to itself. This role is determined due to the relations that the user type has to other objects. For example, in an application process, a relation role would allow access to only those applications to which the user instance has relations to at run-time..

Context Role

The context role is the type of role that can be added to a user-type and is dependent on the attribute values of the user type at the run-time. For example, in an application process, a user with a context role would be able to access all job offers if he has a specific attribute value set.

Modeling of Roles

As it was mentioned, the roles belong to a specific user type or a relation with a user type as target. After the selection of one of those objects, the split view opens and the modeler can add a new role in a special input menu that is depicted in Figure 5.2. The modeler can add various roles to an object. For every role he is able to specify a name and an expression. Because each role is part of a specific object, the modeler can create a role as soon as he has created the object it goes with. This procedure follows the natural thought process and is comprehensible without much explanation. Also, because roles are ordered due to their affiliation with an object, they are not as confusing as the table in the initial concept.

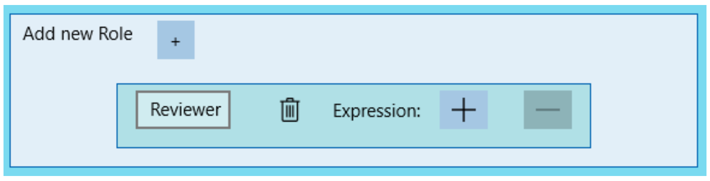


Figure 5.2.: Input option for roles

5.2.2. Permissions

Permissions are created while modeling the micro process, because in this process the user can model each state an object can have and has access to all attributes at this point in time. Each permission is named after its usage, this cannot be influenced by the user. Furthermore, it contains the information if it is instance specific and can have a data context expression. The flag if a permission is instance specific is set at modeling time and used at run-time to determine whether the permission is valid in general for all instances or only for a specific one. With the data context expression, the validity of the permission can be specified, for example a permission-role can be tied to a specific region, in case of an application process, if the role is set to “HR Germany” members of other HR-departments do not hold the permission.

On the server, each permission is attached to the role to which it belongs. This way, deleting a role will automatically delete all permissions that referred to that role as well and data garbage can be evaded. This procedure has one drawback, permissions cannot be saved when no role exists because they cannot be attached. While modeling the process, an empty permission can be added, but this is a dummy permission. The permission will be introduced to the persistence storage of the server after the role was added, if there is no role added every redraw of the model will delete the permission. Also it is not trivial to change a role, if for example the permission has “Role A” set and it is changed to “Role B”, then the application copies the information of “Role A”, requests its deletion and adds the copied information as new permission to “Role B”. The internal code for this procedure can be seen in Listing A.5.

For user experience and easy handling of the permissions they are collected in a permission container to be displayed on the canvas, an example is explained in subsection 5.2.2.

Such a container holds information about the name of the permission, the name of the object (attribute, state type, relation type) the permission belongs to as well as a list of all roles that have this specific permission. It is also important to note, that the permission view does not need special adaptation and can be specified even for overlapping permissions as the system will automatically use the superior permission.

Attribute Read Write Permission

The attribute read write permission specifies, which role is allowed to read or write (which implies the right to read) a specific, non-mandatory attribute in a particular process state. This means that a user without at least the right to read will not see the attribute in the form of the state at run-time, with the read-permission he can see the attribute and its value, but cannot change them, this is only possible with the write permission.

On the server the read write permission is one single type of permission and only distinguished by an enumeration which contains two values, one for read, one for write. However, in the modeling tool it is displayed as two single types of permissions so that on the one hand the user has more overview because he has to deal with two single permissions and on the other hand the created permission container can be reused. This improves the recognition factor of the user, because every permission is in the same container view, only the read write permission would have more than one column and it also supports the software maintenance. If the user defines two overlapping permissions, for example read for “User A” and write for the same user, the system will use write and overwrite the read permission with it because it is the superior permission.

Backward Jump Permission

The backward jump permission gives a user the possibility to jump back to an earlier state of the process. Therefore, it is set to a backward transition in the micro process model.

State Execution Permission

A state can require different attributes as mandatory attributes, look at Section 3.3.1 for further reference. The state execution permission is set to a specific state and grants the specified permission owners a write permission for these attributes, that are referred to by the state.

Transition Commit Permission

After finishing the input form for all attributes of a state the user can exit the state and move to the next state if he holds the transition commit permission.

Further Permissions

More permissions exist within the PHILharmonicFlows concept. However, they are not yet implemented in the application but will be introduced in the future. This concerns the following permissions.

- Object Instantiation Permission
- Object Delete Permission
- Black-Box Activity Execution Permission
- Micro Process Skipping Permission

Even if they are not implemented the concept can be extended without problems.

Modeling of Permissions

The modeling of a permission follows the same principle as the modeling of a role. The permissions are not listed in a table but they are attached to the different objects they belong to. The state execution permission is attached to a state type, the transition commit permission is attached to the outgoing transitions of a state, the backward jump permission is part of the backward transition type and the attribute read write permission can be accessed by selecting a state, after the selection the permissions are shown in the split view.

For each permission, the same options can be adapted. Therefore, all permissions use the same input box for their options mentioned in the previous subsection. Figure 5.3 shows a state execution permission as an example

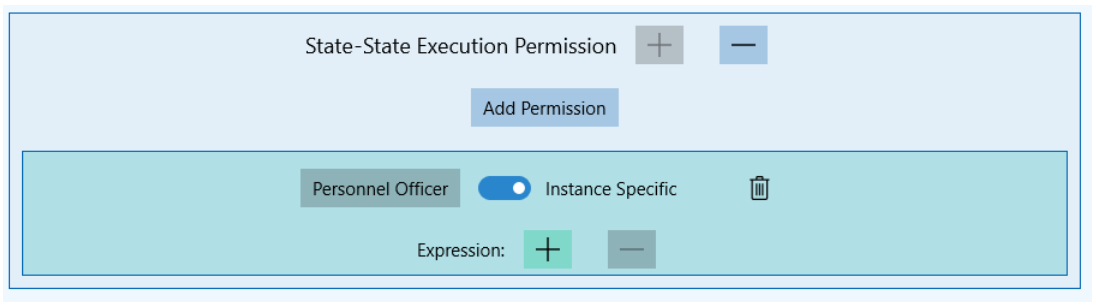


Figure 5.3.: Input option for permission

.

On the top there is a textual description of the permission so a user can instantly identify which permission he is about to model. This description is followed by two buttons that are used to show or hide the detailed permissions. If the permissions are hidden, the overview is better because the user only has to show the details if he want to change them, the hidden permission can be seen in Figure 5.4. These elements are followed by a button that is used to add more permissions, because as previously stated each single permission can only belong to one role but it is possible for many roles to hold the same permission. To guarantee overview for the modeler they all have to be added one by one.

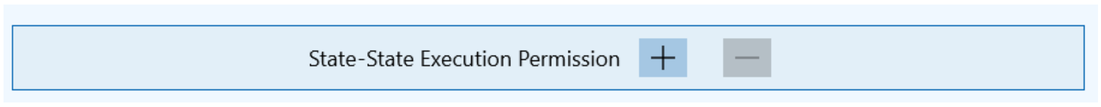


Figure 5.4.: Input options with hidden permissions

For each single permission, that is depicted in the green-blue box the first input is the role that holds the permission, followed by a switch that allows the specification whether the permission is instance specific or not.

In the case that the permission is instance specific the role only holds the permission for the object to which the permission can be applied. If the permission is not instance specific the role holds the permission on all other objects too. For example, in an application process, a case worker that is connected to an object has the role and permission of a personal officer. In this case, he has special instance-specific rights on the job offer he is connected to and all non-instance-specific permissions for all other job offers. For example, he could read all job offers but only write data for the one he is connected to.

The last option that can be added is a data context expression. This expression allows to specify the permission more detailed based on the attributes of the referred object. For example, it is possible to grant the permission only for objects which meet the criteria of the expression. In the exemplary application process it would be possible to grant a role a specific permission only for applications of people that have specified surname. This can be used in the HR-department. In here it is possible to grant the personal officer permission to a caseworker but only for applications with surnames starting with a letter between “A” and “F”. This allows to sort jurisdictions alphabetically as it is often done in government offices.

6 Process Verification

An important part of the modeling tool is that a modeled process can be verified live, while still modeling. This section introduces the course of the process verification, the last key feature. First Section 6.1 illustrates the concept for the verification. In the following Section (6.2) an overview of correctness-criteria for different process parts and an analysis of different types of failures as well as when to do the model-checking is given. Section 6.5 illustrates working example of the verification of state types within the application.

6.1. Verification Concept

The requirement for the concept is that it is extendable, and that it allows a complete model verification as well as the verification of composed or single parts. Therefore, a framework was designed that does a topological sorting of the parts and provides verification methods for each of those parts. Figure 6.1 shows how the different parts of the whole data model are sorted.

The hierarchy depicts the order in which a verification has to be done, because a higher level element cannot be verified without verifying the lower level element or elements first. This means the verification process for such a higher level component is always to verify all of the lower level components that are available in addition to the verification of the actual component.

Furthermore, it is important to know, that elements without a hierarchical relation between them can be verified in parallel if a higher level element is verified or if only a lower level element is verified the parts on the same level are not affected and will be not verified because it is not needed. For example, if a micro process is verified, the process starts with checking the value steps first, then all micro steps, followed by the micro states and at last the micro process itself. The attributes and the macro process are not verified. If, in comparison to that, the object type is verified, then the verification has to be done for attributes, micro process and macro process but those single components can run in parallel.

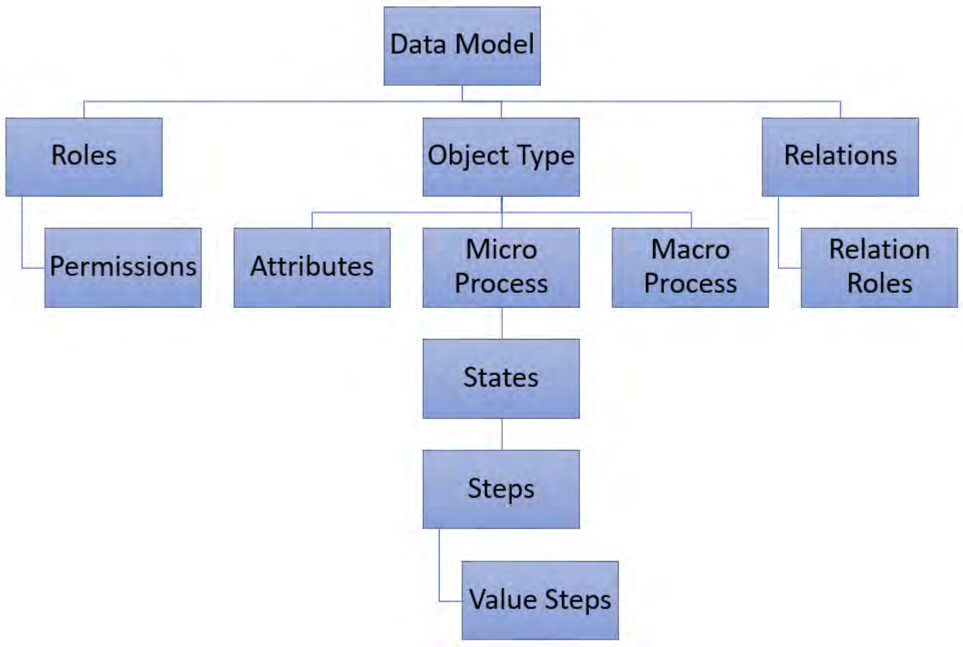


Figure 6.1.: Topological Sorting of Data Model Parts

6.2. Verification Analysis

The verification process is a very complex process because of the many parts that have to be checked. In order to get an overview some verification process characteristics have to be discussed first.

6.2.1. Verification by Scale

Verification can be done on different scales. As it was shown in the concept (Section 6.1) the verification process has different levels and those levels cannot be verified at the same time. To understand the differences, look at the analysis of the points in time. The different scales are needed because there are some failures that can only be checked on a higher level and they can only be checked after all the different sub-elements are checked. For better understanding, there are some examples for the verification process on different levels.

- Data Model
  - The data model verification has to check for possible dead locks within the macro processes, because it is possible that macro processes from different object types block the execution of each other. This problem can only be

found when looking at all macro processes at the same time, not by verifying a single macro process.

- Permissions
  - Each Attribute has to have at least one write permission. The same is true for all other permissions. This can be checked while the micro process is verified but not at verifying a single permission.
- Value Steps
  - Each expression of a value step has to be compatible with the domain of an attribute. Because the value step is the lowest level its verification is not dependent on a higher level component.

6.2.2. Points in Time for Verification

There are different points in time when a verification of the process can be executed.

Live at Modeling Time

The verification can be done live at modeling time. This is used especially for failures that are prohibited at all times by the application itself. While the user is modeling, the system will evaluate his actions and verify them live, if it is prohibited the action will not be completed.

After finishing Modeling

The verification has to be done after the whole model or a specific part of the model is finished. There are two reasons for this classification. The first is that most users feel disturbed if the system warns them about things they have not finished yet but they will complete in the future. For example, if a user creates a new expression, that particular expression will be empty and, as a result of that, not valid. If the system would detect this at modeling time and an error would be shown even if the expression could be corrected in the future. The error would be there until the user finishes the expression and the user might start to get annoyed.

The second reason is that there are some verification actions that consume a lot of time and it is needless to do them at every step. For example, the property that a graph has to be acyclic. While this property is very important it is still a very time consuming thing to check. If this would be checked every time the user draws an edge, the server would either be overloaded or have to be over-sized, which would cost more money for a server that is not used often. Because of that, the best point in time to check for cycles is after the modeling is done, then it is a one-time action.

Before Deployment

The most important time for the validation is right before the deployment. Although normally each part is already verified and the whole model is correct, it is possible that a user changes some parts later and forgets to verify all parts again. For example, if the user changes a micro process, checks it and gets the feedback that the particular process he worked on is correct, he might not think about checking the data model. This could lead to a severe problem, if the user modeled a deadlock between different macro processes. Such a failure could lead to problems at run-time so that processes cannot be instantiated or, in the worst case, the server cannot process the input and comes to a halt because of the error. This is why it is mandatory for the application, that the whole model is checked one last time before the deployment.

6.2.3. Resolving Verification Failures

Because verification failures are different, the solving of those has to be different as well. In general, there are three possible actions„ how resolving can be done.

Manual action required

In this case the application is able to determine that there is a failure, but it is not able to solve it automatically, because of the lack of knowledge or because it would solve the problem on a syntactical level but not on a semantic one.

For example, if the user models a cycle, the application can detect it but it cannot solve it. One edge has to be deleted, but the application cannot decide which of them is the right one to delete. The only possible option is to notify the user and mark the cycle, so that the user can decide, which of the marked edges is not correct.

Another example is the property, that the names of states have to be unique. While the application can determine the fact almost instantly and could solve it as well, because if it finds two similar strings it could just add a number which will be incremented for each finding to all name-strings. This would correct the property instantly, because the formal requirements are fulfilled. But it could change the meaning of the whole process, because the user could have done it accidentally. However, after the automatic action he would not notice and might never see his mistake, so that the whole process would be deployed with semantic problems.

Automatically resolved

Some failures can be automatically resolved, in order to lighten the burden on the user in order to get a correct model. For example, there is a property, that all outgoing transitions of a state end step have to be external. While the user is able to manipulate this and set the value, internally the server would evaluate this and reset the value because it is demanded by the concept. This system behavior is desired because the complete syntactical properties are hard to understand for the common user and it is necessary that the user is guided in his modeling actions.

Automatically resolved issues are sent to the server, saved there and corrected if a verification component is executed.

Instant prevention

If an action is obviously wrong, it will be instantly prevented, because the application simply does not allow the action to be completed. This type of failure is never caused because it is not possible to model it in the first place. It is important to note that with instant prevention the formulated requirement concerning the "Correctness by Construction" (REQ-7) is satisfied.

An example for this is if the user tries to drop a state within another state. This action will never be completed, instead the application shows an implicit warning and ignores the action . This behavior can be seen in figure 6.2.

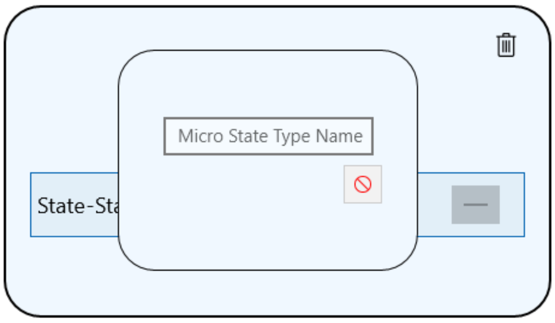


Figure 6.2.: Implicit Error-Message by trying to drop State in State

6.3. Property Analysis

As it was shown in the concept (Section 6.1), every part of PHILharmonicFlows has its own verification process and different properties exist, for all the processes parts. This section shall give a brief overview over the different properties. Most of the properties that have to be met, are mentioned in the original concept of PHILharmonicFlows [11], others were discovered while the application was implemented. Although the criteria are known, there is no assessment when to check the single points in a way that the user does not feel disturbed in his modeling action. Such an evaluation is listed in this section.

Data Model

Besides the verification of the whole data model after completing the modeling, the data model itself has some properties that must be verified. These properties are listed in Table 6.1.

DESCRIPTION OF THE PROPERTY	TIME OF THE VERIFICATION	METHOD OF RESOLVING
the data model has to be acyclic	after modeling	manual action
there are no elements that are not connected to the graph	after modeling	manual action
transitions cannot have the same source and target	checked live	automatic resolving
all object types and user types have to have distinct names	after modeling	manual action
source and target of an edge are distinct	checked live	instant prevention
no elements are boxed into another	checked live	instant prevention
all object types and user types have to have the defined properties set to an attribute	after modeling	manual action
all attributes within an object type must have unique names	after modeling	manual action

Table 6.1.: Property analysis for the data model

Micro Process

Because the micro process is the most complicated model it has the most properties, those are listed in Table 6.3.

DESCRIPTION OF THE PROPERTY	TIME OF THE VERIFICATION	METHOD OF RESOLVING
a micro process has at least two states (one start state and one end state) [1]	after modeling	manual action
a process has exactly one start and at least one end state	after modeling	automatic resolving

[1]The first part of the property is done initially by the system, because the start step will be automatically when the user starts modeling with a new micro process model. However, the user is able to delete this micro step, due to that it is checked after finishing the modeling in every case.

DESCRIPTION OF THE PROPERTY	TIME OF THE VERIFICATION	METHOD OF RESOLVING
a process start state has no incoming transitions [2]	trivial	trivial
a process end step has no outgoing transitions and is an empty micro step [3]	trivial	trivial
each state besides the end state has at least one non-empty micro step	after modeling	manual action
an end state contains at least one micro step and no further micro steps	after modeling	manual action
the process start step is unique and must be an empty micro step	after modeling	manual action
an explicit transition is required to be external	after modeling	automatic resolving
if a micro step has multiple external transitions, they need to be explicit	after modeling	automatic resolving
a micro step can have multiple incoming or outgoing transitions[4]	trivial	trivial
all outgoing internal transitions of a micro step must have distinct priorities	after modeling	manual action
distinct micro steps, which are in the same state, must not have a reference to the same attribute	checked live	instant prevention
value specific micro steps must have at least one value step[5]	trivial	trivial
each value step must have at least one outgoing transition	after modeling	manual action
value-specific micro steps can have outgoing transitions themselves[6]	trivial	trivial
value steps cannot have incoming transitions	checked live	instant prevention
a state can only have backward transitions to a predecessor state	after modeling	manual action
backward transitions cannot have the same state as source and target	checked live	instant prevention
micro steps and value steps can be topological sorted - the process graph is acyclic	after modeling	manual action
all micro steps must belong to a state	checked live	instant prevention

[2]The start state is determined by the system on that particular property.

[3]The end state is determined by the system on that particular property.

[4]Implemented in the design of the application because each vertex can have incoming and outgoing transitions connect to itself.

[5]That is the way a value-specific micro step is defined. Also the application does not distinguish between value-specific or not-value-specific that is done on the sever.

[6]Implemented in the design of the application because each vertex can have outgoing transitions and it is not prevented to do so at a value-specific micro step.

DESCRIPTION OF THE PROPERTY	TIME OF THE VERIFICATION	METHOD OF RESOLVING
external transitions originating from the same source step must be explicit and their target states must be distinct	after modeling	automatic resolving
each micro step must be on a path from the start step to an end step of the micro process	after modeling	manual action
backward transitions between two states are unique	checked live	instant prevention
start states do not have outgoing backward transitions	checked live	instant prevention
end states do not have incoming backward transitions	checked live	instant prevention
all incoming transitions of a step must either be external or internal. No mixtures allowed	checked live	instant prevention
aggregated outgoing transitions are either external or internal	checked live	automatic resolving
the outgoing transitions of a state and end-step are external	checked live	automatic resolving
states cannot contain other states and must not overlap[7]	trivial	trivial
all external outgoing transitions of a micro step are either implicit or explicit, no mixtures allowed[8]	trivial	trivial
all outgoing transitions of a step must either be external or internal, no mixtures allowed[9]	trivial	trivial
a state has exactly one state start step	after modeling	manual action
the process start step must belong to the start state of the process[10]	trivial	trivial
all aggregated outgoing transitions of value-specific micro steps or value steps must have distinct priorities	after modeling	manual action
a state start step only has external incoming transitions[11]	trivial	trivial
a state must have a unique name within the micro process	after modeling	manual action

[7]Implemented in the design of the application because the application prevents dropping of one state into another state, also states cannot overlapped because of automatic placement, look at Section 3.3.3 for further reference.

[8]External outgoing transitions have only those two options. Per default they are implicit, but the modeler can change this.

[9]Only those two options exist. Changes depend on the created model and are determined by the system.

[10]The start step is determined by the server based on this property.

[11]The start step is determined by the server based on this property.

DESCRIPTION OF THE PROPERTY	TIME OF THE VERIFICATION	METHOD OF RESOLVING
source and target of a micro step must be distinct	checked live	instant prevention

Table 6.3.: Property analysis for the micro process

Macro Process

The macro process might be the model that is the easiest to understand because it only has two types of elements, yet it still has some properties that have to be fulfilled, those are listed in Table 6.4.

DESCRIPTION OF THE PROPERTY	TIME OF THE VERIFICATION	METHOD OF RESOLVING
data model has to be acyclic	after modeling	manual action
the start macro step has to refer to the start state	after modeling	manual action
the end macro step has to refer to an end state	after modeling	manual action
macro transitions cannot have the same macro step as source and the same port as target	checked live	instant prevention

Table 6.4.: Property analysis for the micro process

Expression

While each expression has to be correct for the context in which it is used, there are some general properties that each expression has to fulfill. First, expressions have to be complete, this is checked after the expression is modeled and the failure that could arise from it requires manual action, because the system cannot decide how to finish the expression and there is no saving of partly correct expressions.

Second, used operators have to be valid for the return type of its child, this is checked after modeling and requires manual action. Because this property is very important, figure 6.3 illustrates it. While both expressions (cf. Fig. 6.3 (1),(2)) in the left as well as in the right sub-tree are correct, the binding root-expression (cf. Fig. 6.3 (3)) is incorrect because the left-side expression (cf. Fig. 6.3 (1)) returns a boolean and the right-side expression (cf. Fig. 6.3 (2)) returns a number and both values cannot be summed up. In Table 6.5 is a complete set of valid operators and return types for a binary expression.

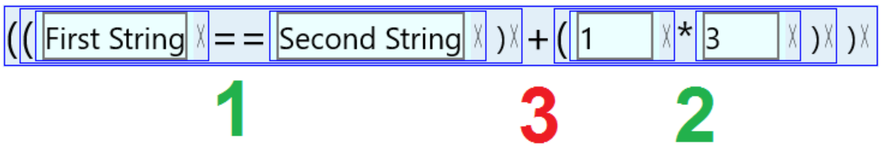


Figure 6.3.: Expression with correct and wrong used Operators

	OPERATOR				
	Compare Difference	Equality	Logic	Addition	Multiplication
Input:	(>,>=,<=,<)	(==,!=)	(||,&&,^,->)	(+,-)	(*,/,%)
Boolean	no	yes	yes	no	no
Date	yes	yes	no	no	no
Number	yes	yes	no	yes	yes
String	no	yes	no	no	no
Return Type:	Boolean	Boolean	Boolean	Number	Number

Table 6.5.: Overview of operator usage and return-types

General

All general properties have to be checked before the completed model is deployed on the server, as it was mentioned above in Section 6.2.2. So far there is only only one property discovered. This property is that different macro processes cannot have conflicting details that would lead to a deadlock. As previously mentioned, solving this error needs manual action.

6.4. Reporting of Errors

While the process verification is a very important part for the user, as well as the application, the actual reporting of errors might be the more important thing for the user. If a verification fails, that cannot be automatically resolved, the user has to be notified in a clear and understandable way, so that the failure is understandable and can be corrected by the user. Only if both of these conditions are met, the resolution will be successful. In order to achieve this, a special reporting framework was built.

6.4.1. Reporting Framework

The framework is based on verification modules for error reporting. In Figure 6.4 an overview can be seen. The super-class of the verification part is the *AbstractVerification-Component* all verification modules are sub-classes of this class. A module is a single verification step. An example for such a module is the class *AllStatesHaveDistinctNames* which is explained in detail in Section 6.5. The subclass contains information about the name and the dependencies, that can be used for a complete verification of greater parts of the application as it is described in Section 6.1 and an verification component identifier.

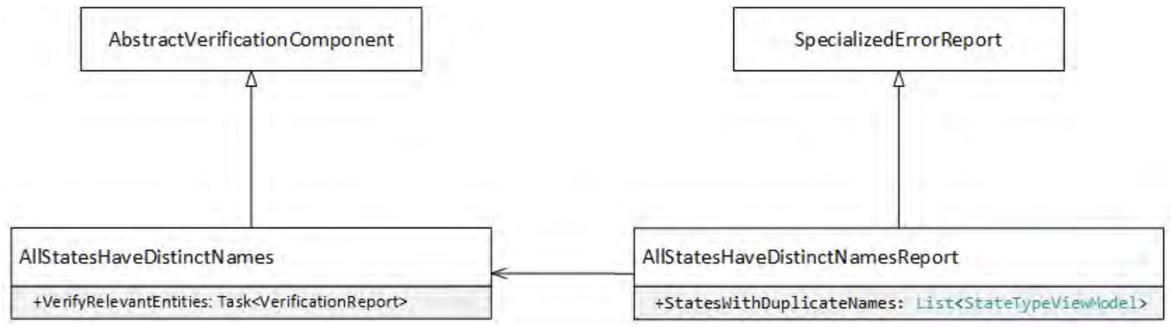

AbstractVerificationComponent
SpecializedErrorReport
AllStatesHaveDistinctNames
+VerifyRelevantEntities: Task<VerificationReport>
AllStatesHaveDistinctNamesReport
+StatesWithDuplicateNames: List<StateTypeViewModel>

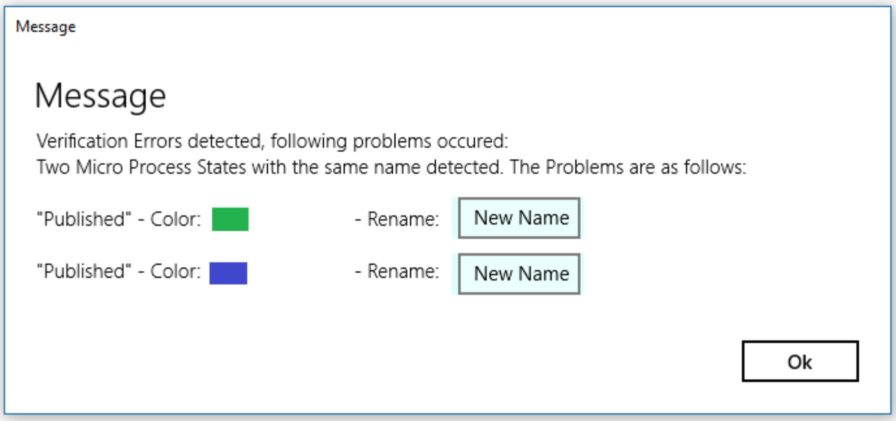
Figure 6.4.: Overview Verification and Reporting Framework

Each sub-class has its own verification component identifier, which is a special enumeration, to list and identify all possible verification modules. The verification itself is done in a special method, that is called *VerifyRelevantEntities* and has to be implemented by all sub-classes.

To report errors after they were identified, a specialized report is created for each verification method, in the example this is done by the class *AllStatesHaveDistinctNamesReport*. It comes with a description, an error message and the component identifier. Apart from that every subclass can contain further information as necessary.

6.4.2. Extension possibilities in the future

In the future this concept can be extended. It is possible that the system would allow the user to instantly input the correction, without searching for the failure first and correct it manually in the model. A possible example for this would be the “unique state name” property. In this case the dialogue could show the duplicate names, the color in which the different states are highlighted for a quick overview and an option to insert a new name for all states that were detected by the verification. Figure 6.5 shows a mock-up of the concept.


Message

Message

Verification Errors detected, following problems occured:
Two Micro Process States with the same name detected. The Problems are as follows:

"Published" - Color: 🟩 - Rename: New Name

"Published" - Color: 🟦 - Rename: New Name

Figure 6.5.: Conceptual Idea of an Advanced Message Dialogue

6.5. Example usage

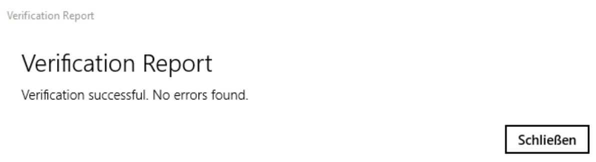
Because the framework is only theoretical at the moment and the verification process itself is very difficult and might not be completely implemented while the application is still under development most of the work is done theoretically by analyzing properties. But for some modules there is an implementation to demonstrate the process in general and prove that the concept can be implemented, that it works in a real application and that it will satisfy the requirement for a “Simple Verification Concept” (REQ-11).

6.5.1. Property - Unique State Names

One verification that is implemented is the verification of the property that all states have to have distinct names. The class that was created for that reason is named *AllStatesHaveDistinctNames*. This class extends the abstract super-class and gets its own component identifier. It also implements the verification method. This method is shown in Listing A.6. It checks all states of a micro process and saves the view models of all duplicates that are found.

After this, either no duplicates were found and the verification was successful, or a specialized error report will be returned. This error report is called *AllStatesHaveDistinctNamesReport* and extends the abstract class *SpecializedErrorReport*. In addition to the inherited properties, the subclass also contains a list of view models of all states that are found to be duplicates. This list is used to mark errors, that the verification found, for the user. Figure 6.6 shows a verification that found no errors.

If the verification does find an error, this error is marked visually in a signaling color for the user. In this case the first and the second micro state have an equal name, “Published” in this example. The states are marked in orange-red (cf. Appendix C,

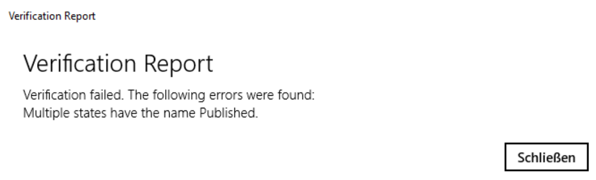

Verification Report

Verification Report

Verification successful. No errors found.

Figure 6.6.: Correct verification

Table C.1 (“Background Brush XII”)) and a message shows the user the problem. The message dialogue is depicted in Figure 6.7. The marking is done by the application, because in the case of an error all graphical objects of a special type, in this case the *MicroStateTypeView*, are checked. If they have the same id as the view models that are returned by the verification framework a method is called that marks the state in a special color. The verification can be executed by the user by clicking on the verify button in the lower left corner of the graph window (cf. Appendix B, Fig. B.2 (4)). Listing A.7 shows how the verification can be executed and in Listing A.8 it is shown how the application processes the results.


Verification Report

Verification Report

Verification failed. The following errors were found:
Multiple states have the name Published.

Figure 6.7.: Verification with Problem and Marking of the Errors

7 Related work

While PHILharmonicFlows is a good approach in the world of process management systems of course there are other approaches as well. This section is used to show some of those approaches and discuss them to point out similarities and differences.

7.1. Artifact Centric - Barcelona

In the year 2010 the Guard-Stage-Milestone (GSM) approach for the modeling of business artifact life cycles was introduced by IBM researcher Richard Hull and his colleagues. With this approach, stakeholders should be more involved with the managing of business operations. Furthermore, rapid prototyping of Business Operation Models and unlimited support of other prescriptive managing approaches should be enabled with GSM ([9, 7, 20]).

For this approach different parts were introduced. First a text-based programming language called GSM-L and second a prototype engine, named Barcelona. This particular modeling tool should be compared with the modeling tool of PHILharmonicFlows.

Overview of Business Artifacts

Business artifacts are composed of two parts. The first one is the information model. This model describes the relevant data for that is used for the whole process. The second part is the life-cycle model that describes how the information model is affected and how the different factions involved in the process, like stakeholders, can interact with each other. Different approaches can be used to describe a life-cycle model, in Barcelona[1], these are a finite state machine life-cycle and a declarative life-cycle. The last one is the GSM life-cycle and is used mainly.

[1]Available as freeware on the following link: https://sourceforge.net/projects/bizartifact/?source=dlp

Modeling tool Barcelona in detail

The Barcelona modeling tool comes with a execution engine, a solution designer editor, a default run time GUI and some additional components that comprise repositories and external services like REST and WDSL. Figure 7.1 shows an overview of the architecture.

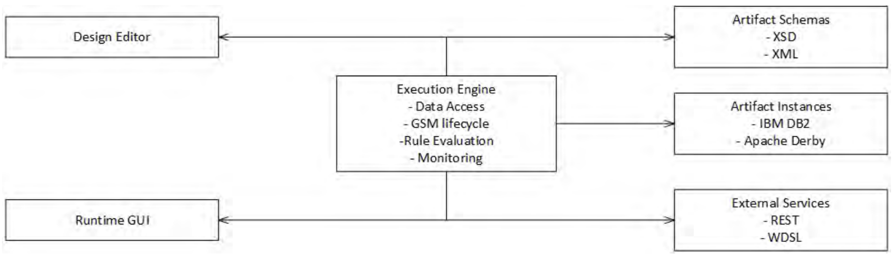
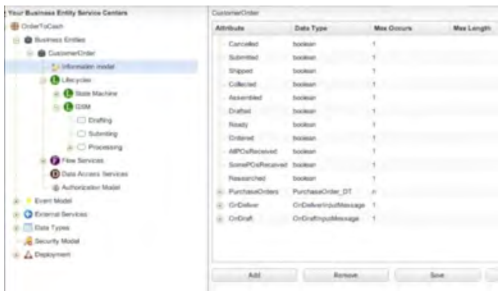

Design Editor
Artifact Schemas - XSD - XML
Execution Engine - Data Access - GSM lifecycle -Rule Evaluation - Monitoring
Artifact Instances - IBM DB2 - Apache Derby
Runtime GUI
External Services - REST - WDSL

Figure 7.1.: Barcelona architecture ([8])

To discuss Barcelona in more detail the demo that was created for the International Conference on Service Oriented Computing 2013 (ISOC 2013) and a Seminar of the University of Rome were used ([8, 29, 28]).

Barcelona supports different artifacts, which are embedded in an environment and can send and receive messages to and from this environment. All attributes are stored in the information model. In contrast to PHILharmonicFlows this attributes are not as ordered and understandable because they can be simple or complex and are all shown in a single list as it is depicted in Figure 7.2.



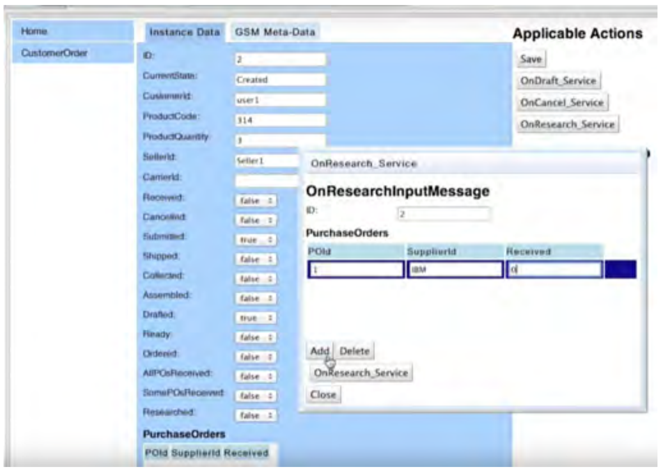
CustomerOrder

Attribute	Data Type	Max Occurs	Max Length
Cancelled	boolean	1	
Submitted	boolean	1	
Shipped	boolean	1	
Collected	boolean	1	
Assembled	boolean	1	
Drafted	boolean	1	
Ready	boolean	1	
Ordered	boolean	1	
AllPOsReceived	boolean	1	
SomePOsReceived	boolean	1	
Researched	boolean	1	
PurchaseOrders	PurchaseOrder_DT	n	
OnDeliver	OnDeliverInputMessage	1	
OnDraft	OnDraftInputMessage	1	

Figure 7.2.: Attributes in the information model

While it is beneficial to the user that he can see all available objects in a tree-view it can also be confusing, because of the information overload. Nonetheless this feature might have to be introduced to the PHILharmonicFlows application as well so that the modeler can have all modeled elements on one single view.

Furthermore it is possible to model the life-cycle model. Compared to the PHILharmonicFlows framework, the modeling in Barcelona looks rather old fashioned and not very intuitive. In the demo the process starts with drafting, followed by submitting and closes with the processing. This process course is not clear on first sight because in contrary



ID:	2
CurrentState:	Created
CustomerId:	user1
ProductCode:	314
ProductQuantity:	3
SellerId:	Seller1
CarrierId:	
Received:	false
Cancelled:	false
Submitted:	true
Shipped:	false
Collected:	false
Assembled:	false
Drafted:	true
Ready:	false
Ordered:	false
AllPOsReceived:	false
SomePOsReceived:	false
Researched:	false

PurchaseOrders

POId	SupplierId	Received

OnResearchInputMessage

ID: 2

PurchaseOrders

POId	SupplierId	Received
1	IBM	d

Figure 7.3.: Form in the Barcelona demo

to the PHILharmonicFlows application, there are no edges which a modeler can use to follow the process easily.

All specification is done via forms. While this is possible and working it is not as natural as building the process with edges and vertexes and thereby being able to get a graphical representation for the whole process.

Summarizing, the approach taken by GSM and Barcelona could be a good one but the application is not comparable to the one created for PHILharmnonicFlows. First it does not support multiple devices like tablets, second it is not very intuitive and third it looks rather old fashioned. An example for this can be seen in Figure 7.3, that shows a form in Barcelona.

7.2. Activity-centric business process modeling - BPMN 2.0

The activity-centric modeling in form of Business Process Modeling (BPMN 2.0) is “the culmination of two streams of work from the late 1990’s and early in the this century” ([31]). It was made public in 2004 and is nowadays maintained by the Object Management Group ([3]). BPMN 2.0 supports process modeling with different elements ([4]).

- Start and End events
  - used at the beginning and the end of each modeled process
- Activities
  - used to describe the activity that has to be done in a particular step of the process
- Gateways

  - used to get some additional logic into the course of the process
  - gateways allow the user to model an “AND”, a “XOR” an “OR” as well as some events
- Pools and Lanes
  - used to specify competences of people that are involved in the process
- Messages
  - used to ensure communication in the process

In comparison to PHILharmonicFlows BPMN2.0 is simpler, due to the fact that a process can be described with a single collaboration diagram. This enables the user to understand it faster, but it also less powerful due to the fact that processes do not possess automatic generated variants and forms.

Modeling BPMN 2.0

Due to the simplicity BPMN 2.0 can be modeled in various tools. For example Microsoft Visio[2]. This is an application from Microsoft that enables diagramming and vector graphics. BPMN 2.0 models can be drawn by dragging the elements on the drawing-board and connecting this. While this version of modeling is akin to the application built for this master’s thesis and does work without a server or any other infrastructure it has also a major drawback. It is not possible to verify it. However, Visio grants the user more possibilities because it is not limited to a single model, but supports almost every possible model.

And while there are other tools, that enable to verify a built process there are no tools that come with a run-time environment that is able to execute instances of the modeled process, like it is possible in PHILharmonicFlows.

[2]https://products.office.com/de-de/visio/flowchart-software

8 Summary and Outlook

While working on this master's thesis, a modeling tool for PHILharmonicFlows was designed and implemented. In Section 2, there is an explanation of the fundamental parts of the concept. Followed by an overall introduction to the modeling in Section 3, there are all the possibilities the tool has explained. After that in the Sections 4 to 6 there is a detailed introduction of the key features the application possesses.

First is the expression modeler, that allows the user to freely model an expression of his choice. While it allows as much freedom as possible, at the same time it guides and supports the user, to ensure that variables are used correctly while no wrong references are set. Due to its importance, this part of the tool was very time consuming and worked over twice due to some adaptations and the possibility to support n-ary expressions at last. Also it was time consuming, the final result is convincing and has potential because it can be extended freely.

The second feature is the overhaul of the roles and permissions that are used within the framework. Within the context of this part of the development, the confusing and user-unfriendly table, that was introduced in the original concept was redesigned. Instead, the roles and permissions were added to the objects they belong to and where a user would expect them. This way, the modeling workflow was significantly improved.

As a last key feature, the process verification was analyzed and an example was developed that shows how an automated verification can be done and how found problems can be highlighted in an understandable fashion for the user.

The thesis closes with an overview of related work. Therefore Section 7 introduces other possibilities how a process can be modeled in an application.

To realize the modeling tool, the Universal Windows Platform and Visual Studio 2015 was used. This combination was a very good choice because it not only supports the application development due to an intuitive IDE and very good IntelliSense but it also came with a very good documentation and some helpful videos for training. Some features are still missing, like the adaptation of code while debugging, but they are planned to be functional in future releases of the IDE.

In further development stages of the modeling tool and the whole PHILharmonicFlows-Framework there are some steps to take. First of all, the modeling tool has to be used to model more processes in order to identify and eliminate shortcomings that are not taken into consideration by the concept. Only then it is guaranteed that the framework is able to model each possible situation of a business process.

As a second step the modeling tool has to be extended in order to work properly on tablets and as a web-solution. While those steps are prepared and taken into consideration by using the Windows Universal Platform as a framework, there are some adaptations necessary. For example, some input controls have to be adapted for the use on a tablet where it is not as comfortable to drag and drop items as it is with the mouse. Instead it would be possible to use a click-choice menu. Besides the input that could be done with forms on websites it could be also interesting to extend the modeling tool in order to allow web-based modeling. This could be interesting for enterprises with the need of modeling various processes relatively fast. Additionally, in this scenario it would be possible to use PHILharmonicFlows for educational purposes in universities or colleges, because with a web-based modeling the application can run on any device or operating system with the same overall conditions. Furthermore, due to the verification process that can be highly specified students that work with PHILharmonicFlows do not need much attention of tutors, because the verification as well as the error reporting is mostly self-explanatory.

As a third step the application has to be capable to fully support variants and schema evolution. This way, it will be possible to change and migrate models. This will allow PHIL-harmonicFlows to be more flexible and therefore improve its productive use, because variants, as well as schema evolution, are very common in real world processes. With the planned design of both aspects, it will be possible to adapt processes live at run-time, this will give an enterprise great flexibility without loosing the safety of documentation and machine-readable processes, which come with the use of a process management tool. This will increase the usability as well as the value of the framework.

As a last step, the whole solution can be extended with Business Intelligence features. For enterprises, it is necessary to have a feedback-loop in order to optimize the course of their processes. Therefore, it is handy to have the opportunity to add key performance indicators to a process in order to analyze it later. An example for this would be the measurement of time in an order process. If an enterprise orders some materials they often are payed automatically if the invoice number and sum matches the information on the order. Now each instance of the modeled process could carry the information on how long it took from ordering to payment. This way, problems can be identified and solved. Besides the possibility to model those steps, an overview, like a dashboard, would also be needed in order to present the gathered information to an analyst.

A
Sources

```
private void OperandOnExpViewDeleted(object sender,
   GeneralExpEventArgs generalExpViewEventArgs)
{
   if (generalExpViewEventArgs.Handled == false)
   {
      if (generalExpViewEventArgs.GeneralExpView != this)
      {
         generalExpViewEventArgs.Handled = true;
         for (int index = 0; index < OperatorAndArity.Children.
            Count(); index++)
         {
            if (OperatorAndArity.Children[index] ==
               generalExpViewEventArgs.GeneralExpView)
            {
               OperatorAndArity.Children.RemoveAt(index);
               ExpView expresion = new ExpView(null, this,
                  _attributeList ?? null);
               expresion.ExpViewDeleted +=
                  OperandOnExpViewDeleted;
               OperatorAndArity.Children.Insert(index,
                  expresion);
            }
         }
      }
   }
}
```

Listing A.1: Handling of events from nested classes

```xml
<Grid Background="{StaticResource ExpViewBackground}" MinWidth=
   "20" MinHeight="20">
   <StackPanel Orientation="Horizontal">
      <Grid x:Name="ExpContainer">
         <StackPanel x:Name="OperatorAndArity" Orientation="
            Horizontal">
            <Grid x:Name="ExpInit" Background="{StaticResource
               ExpInitGrid}" MinHeight="20" MinWidth="20"
               PointerPressed="ExpInit_Ary_OnPointerPressed">
               <TextBlock x:Name="OptionStatus"
                  VerticalAlignment="Center"
                  HorizontalAlignment="Center">Choose Arity</
                  TextBlock>
            </Grid>
         </StackPanel>
      </Grid>
      <Button x:Name="BtnDelete" Background="{StaticResource
         DeletBtnBrush}" HorizontalAlignment="Stretch" Click="
         BtnDelete_Click">
         <SymbolIcon Symbol="Clear"/>
      </Button>
   </StackPanel>
</Grid>
```

Listing A.2: XAML of the ExpView-Class

```
private void AddMenuFlyoutToAButton(Button btn, List<string>
   flyoutItemList)
{
   MenuFlyout mf = new MenuFlyout();
   foreach (string word in flyoutItemList)
   {
      MenuFlyoutItem mfi = new MenuFlyoutItem();
      mfi.Text = word;
      mfi.Click += delegate
      {
         btn.Content = mfi.Text;
         _nullaryExpression.AttributeTypeId = _attributeList.
            Single(c => c.Name == mfi.Text).ID;
      };
      mf.Items.Add(mfi);
   }
   btn.Flyout = mf;
}
```

Listing A.3: Method to add a flyout menu to a button

```csharp
private void BuildNullaryExpression_Constant_Number() {
   ExpInit.Children.Clear();
   TextBox tb = new TextBox()
   {
      Margin = new Thickness(5, 0, 5, 0),
      VerticalContentAlignment = VerticalAlignment.Center
   };
   string valAsString = _nullaryExpression?.Value.ToString();
   double val;
   if (Double.TryParse(valAsString, out val))
   {
      tb.Text = val.ToString();
   }

   tb.TextChanging += delegate
   {
      if (!Regex.IsMatch(tb.Text, "^\\d*\\.?\\d*$") && tb.Text
         != "")
      {
         int pos = tb.SelectionStart - 1;
         tb.Text = tb.Text.Remove(pos, 1);
         tb.SelectionStart = pos;
      }

   };
   tb.TextChanged += delegate
   {
      _nullaryExpression.Value = Double.Parse(tb.Text);
   };
   tb.LostFocus += delegate
   {
      ExpNullaryGraphicChanged?.Invoke(this, EventArgs.Empty);
   };
   ExpInit.Children.Add(tb);
   ExpNullaryGraphicChanged?.Invoke(this, EventArgs.Empty);
}
```

Listing A.4: Method for the number constant

```
private async void UpdatePermissionOnServer(RoleViewModel role)
{
   if (PermissionViewModel.Role != null && PermissionViewModel.
      Role.ID == role.ID)
   {
      //no Change
      return;
   }
   else
   {
      //Check if the Permission was saved
      if (PermissionViewModel.Role == null)
      {
         //Permission was never saved on the Server -> add a
            new permissison with Role
         PermissionViewModel.Role = role;
         PermissionViewModel = await Global.Client.
            AddPermissionAsync(PermissionViewModel);
      }
      else
      {
         //There is a Permission on the Server -> delete the
            permission and add a new permission with a
            different role
          await Global.Client.DeletePermissionAsync(
             PermissionViewModel);
         PermissionViewModel.ID = null;
         PermissionViewModel.Role = role;
         PermissionViewModel = await Global.Client.
            AddPermissionAsync(PermissionViewModel);
      }
    }
}
```

Listing A.5: Changing a permission role

```csharp
public async Task<VerificationReport> VerifyRelevantEntities(
   MicroProcessType relevantEntity) {
   //... Get States
   for (int ind1 = 0; ind1 < states.Count; ind1++)
   {
      for (int ind2 = (ind1 + 1); ind2 < states.Count; ind2++)
      {
         if (states[ind1].Name == states[ind2].Name)
         {
            StateTypeViewModel stvm1 = await StateTypeViewModel.
               CreateAsync(states[ind1]);
            StateTypeViewModel stvm2 = await StateTypeViewModel.
               CreateAsync(states[ind2]);

            bool found_1 = foundDuplicates.Any(stvm => stvm.ID
               == stvm1.ID);
            if (!found_1)
            {
               foundDuplicates.Add(stvm1);
            }

            bool found_2 = foundDuplicates.Any(stvm => stvm.ID
               == stvm2.ID);
            if (!found_2)
            {
               foundDuplicates.Add(stvm2);
            }
         }
      }
   }

   //... Determine if there were duplicates and generating a
      message in case there were any

      //create a specialized report and return it
      var specializedReport = new
         AllStatesHaveDistinctNamesErrorReport(errorMsg,
         VerificationComponentIdentifier, foundDuplicates);
      return new VerificationReport(false, specializedReport);
   }
}
```

Listing A.6: Verification Method Example

```
private async void EvaluateModelBtn_OnClick(object sender,
   RoutedEventArgs e)
{
   //Redraw the model to remove possible marking colors form
      former verification processes
   await RedrawModel();
   bool verificationResult = true;
   string verificationErrorMsg = "Verification Errors detected,
       following problems occured: \n";

   //Distinct Names in States - Check
   Tuple<bool, string> checkAllStatesHaveDistinctNames = await
      EvaluationCheck_AllStatesHaveDistinctNames();
   if (!checkAllStatesHaveDistinctNames.Item1)
   {
      verificationResult = false;
      verificationErrorMsg += checkAllStatesHaveDistinctNames.
         Item2 + "\n";
   }

   //Further Checks
   //...

   //Summary
   if (!verificationResult) //Error
   {
      await UserDialogExtension.MsgBox(verificationErrorMsg);
   }
   else //No Error
   {
      await UserDialogExtension.MsgBox("Verification
         Successfull. No Problems detected.");
   }
}
```

Listing A.7: Execution of the Verficication

```
private async Task<Tuple<bool,string>>
   EvaluationCheck_AllStatesHaveDistinctNames()
{
  //Verification method called via webservice
   var allStatesHaveDistinctNamesErrorReport = await Global.
      Client.VerifyAllStatesHaveDistinctNamesAsync(
      _objectTypeID);
   //case: no problems found
   if (allStatesHaveDistinctNamesErrorReport == null)
   {
      return new Tuple<bool, string>(true, "");
   }
   else //case: problems detected
   {
      //marking of the problematic graphical objects
      foreach (StateTypeViewModel state in
         allStatesHaveDistinctNamesErrorReport.DuplicatedStates
         )
      {
         var microStateView =GraphGrid.Children.OfType<
            MicroStateTypeView>().FirstOrDefault(x => x.
            StateTypeViewModel.ID == state.ID);
         microStateView?.SetStateErrorMarking();
      }
      //return the error message
      return new Tuple<bool, string>(false,
         allStatesHaveDistinctNamesErrorReport.ErrorMessage);
   }
}
```

Listing A.8: Handling of the verification results

B
Figures

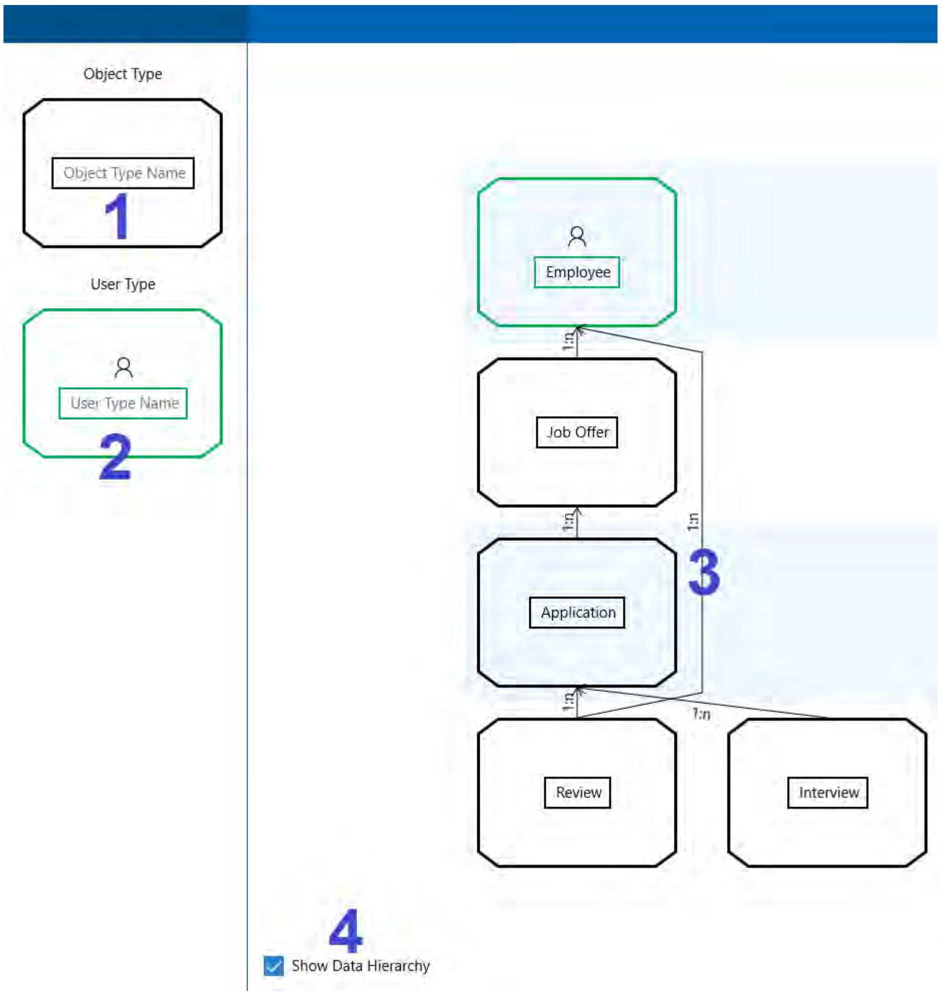


Figure B.1.: Overview Data Model

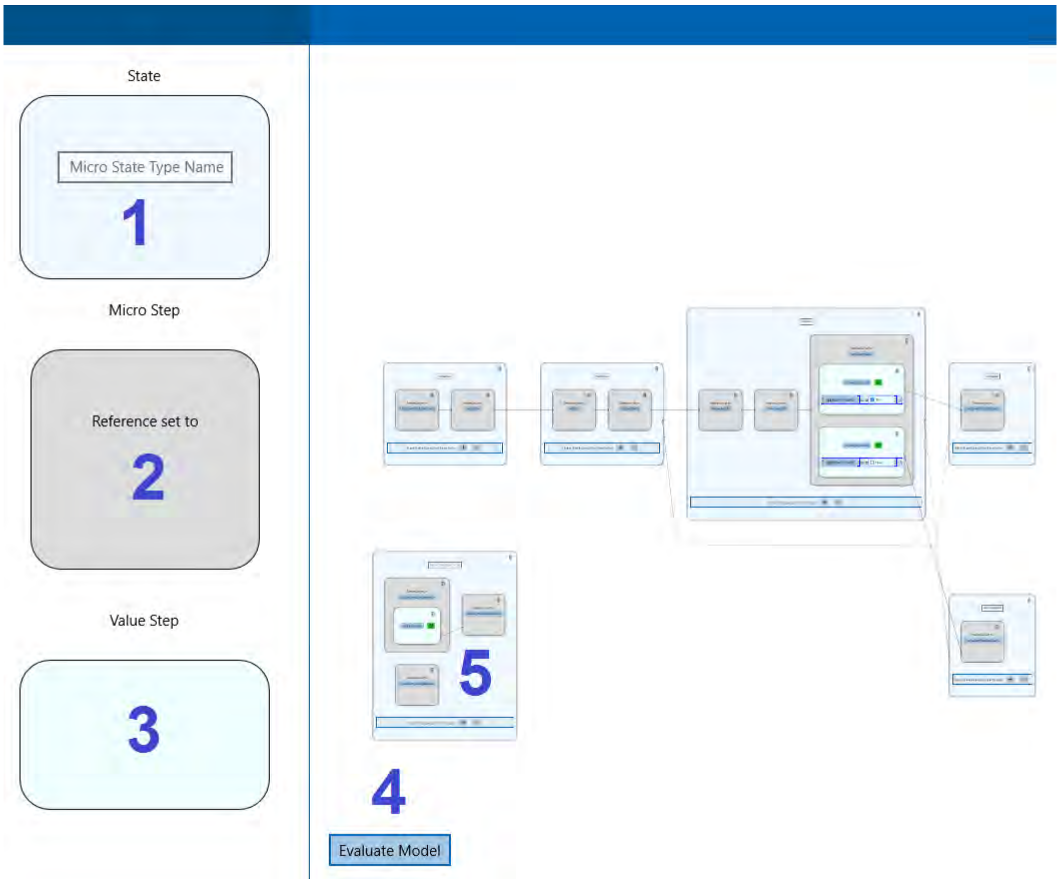


Figure B.2.: Overview Micro Process

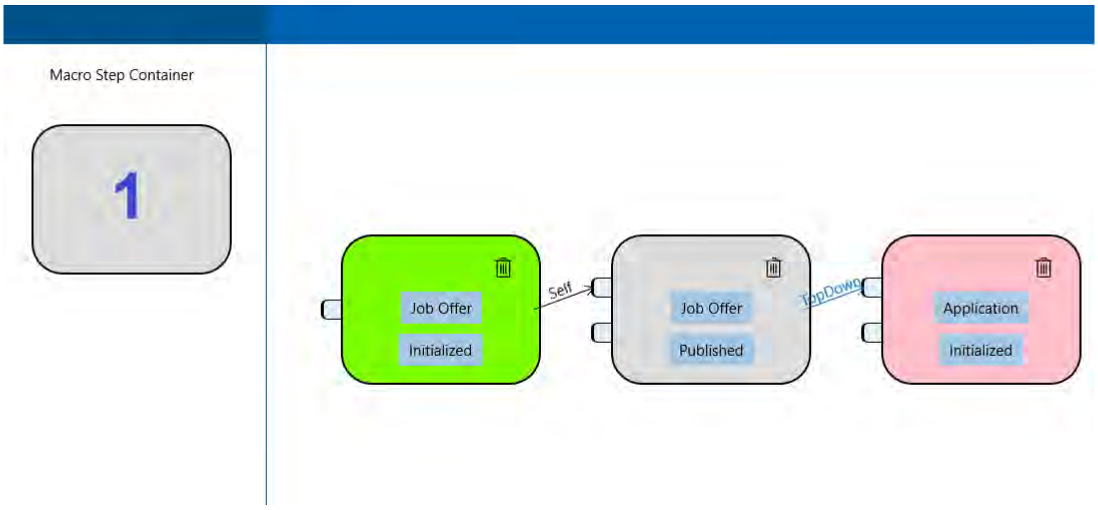


Figure B.3.: Overview Macro Process

C
Tables

USAGE	COLOR
Background Brush I	🟦
Background Brush II	⬜
Background Brush III	⬜
Background Brush IV	🟩
Background Brush V	🟥
Background Brush VI	⬜
Background Brush VII	⬜
Background Brush VIII	🟩
Background Brush IX	⬜
Background Brush X	🟦
Background Brush XI	🟦
Background Brush XII	🟧
Background Brush XIII	🟥
Border Brush I	🟦
Border Brush II	🟩
Border Brush III	⬛
Border Brush IV	🟨
Border Brush V	🟦

Table C.1.: Used Colors

List of Tables

List of Figures

Bibliography

[1] BECK, Hannes: *Implementierung einer Komponente zur Modellierung von Mikro-Prozessen in einem datenorientierten Prozess-Management-System*, Ulm University, Bachelor-Thesis, 2012

[2] BENABBAS, Yassine: *Automated UI testing of a UWP app using Appium*. Website. `https://medium.com/@yostane/automated-ui-testing-of-a-uwp-app-using-appium-dc10d8df6631#.nragbi9ug`. Version: 2016. – last visited on 2016/09/18

[3] BPMNFORUM: *BPMN Frequently Asked Questions*. Website. `http://bpmnforum.com/bpmn-faq/`. – last visited on 2016/10/20

[4] BULLES, John u. a.: *BPMN 2.0 by Example*. Technical Document, 2010

[5] BYKOV, Sergey u. a.: Orleans: A Framework for Cloud Computing. In: *whitepaper* (2010)

[6] COLIN ARMISTEAD, Jean-Philip P. ; MACHIN, Simon: Strategic Business Process Management for Organisational Effectiveness. In: *Long Range Planning, Vol. 32, No. 1, pp. 96 to 106* (1999)

[7] DAMAGGIO, Elio ; HULL, Richard ; VACULÍN, Roman: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard–Stage–Milestone lifecycles. In: *Information Systems* 38 (2011), Nr. 4, S. 561–584

[8] HEATH, Fenno u. a.: Barcelona: A Design and Runtime Environment for Declarative Artifact-Centric BPM. In: *Service-Oriented Computing 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013. Proceedings, pp.705 to 709* (2013)

[9] HULL, Richard u. a.: Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles. In: *Springer-Verlag LNCS 6551.* (2010)

[10] KÜNZLE, Vera: Towards a Framework for Object-aware Process Management. In: *1st Int'l Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11)* (2011)

[11] KÜNZLE, Vera: *Object-Aware Process Management*, University Ulm, Diss., 2013

[12] KÜNZLE, Vera ; REICHERT, Manfred: Herausforderungen auf dem Weg zu datenorientierten Prozess-Management-Systemen. In: *EMISA Forum, 29 (2). pp. 9 to 24* (2009). `http://dbis.eprints.uni-ulm.de/533/`

[13] KÜNZLE, Vera ; REICHERT, Manfred: Integrating Users in Object-aware Process Management Systems: Issues and Challenges. In: *Proceedings BPM'09*

Workshops, 5th Int. Workshop on Business Process Design (BPD'09), Springer, September 2009, S. 29–41

[14] KÜNZLE, Vera ; REICHERT, Manfred: Towards Object-aware Process Management Systems: Issues, Challenges, Benefits. In: *Proc. 10th Int'l Workshop on Business Process Modeling, Development, and Support (BPMDS'09)*, Springer, Juni 2009, S. 197–210

[15] KÜNZLE, Vera ; REICHERT, Manfred: A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In: *Proc. 12th Int'l Working Conference on Business Process Modeling, Development and Support (BPMDS'11)*, Springer, Juni 2011, S. 201–215

[16] KÜNZLE, Vera ; REICHERT, Manfred: PHILharmonicFlows: Research and Design Methodology / University of Ulm. 2011 (UIB-2011-05). – Technical Report

[17] KÜNZLE, Vera ; REICHERT, Manfred: Striving for Object-aware Process Support: How Existing Approaches Fit Together. In: *1st Int'l Symposium on Data-driven Process Discovery and Analysis (SIMPDA'11)*, Springer, 2012, S. 169–188

[18] KÜNZLE, Vera ; WEBER, Barbara ; REICHERT, Manfred: Object-aware Business Processes: Properties, Requirements, Existing Approaches / University of Ulm. Version: August 2010. http://dbis.eprints.uni-ulm.de/681/. Ulm, Germany : University of Ulm, August 2010 (UIB-2010-06). – Technical Report

[19] M. WESKE, W.M.P. van der A. ; VERBEEK, H.M.W.: Guest editorial - Advanceds in business process management. In: *Data & Knowledge, Vol.50, pp. 1 to 8* (2004)

[20] MARIN, Mike A. ; HULL, Richard ; VACULÍN, Roman: Data Centric BPM and the Emerging Case Management Standard: A Short Survey. In: *Business Process Management Workshops*, 2012

[21] MICROSOFT, Corporation: *Partial Classes and Methods*. Website. https://msdn.microsoft.com/en-us/library/wa80x488.aspx. Version: 2016. – last visited on 2016/08/15

[22] MICROSOFT, Corporation: *What Is Windows Communication Foundation*. Website. https://msdn.microsoft.com/de-de/vstudio/aa663324.aspx. Version: 2016. – last visited on 2016/09/04

[23] MICROSOFT CORPORATION, Jim W.: *Calendar, date, and time controls*. Website. https://msdn.microsoft.com/en-us/windows/uwp/controls-and-patterns/date-and-time. Version: 2016. – last visited on 2016/08/19

[24] MICROSOFT CORPORATION, Tyler W.: *Guide to Universal Windows Platform (UWP) apps*. Website. https://msdn.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide. Version: 2016. – last visited on 2016/09/04

[25] MICROSOFT CORPORATION, Vaclav T.: *Introduction to Service Fabric Reliable Actors*. Website. https://azure.microsoft.com/en-gb/documentation/articles/service-fabric-reliable-actors-introduction/. Version: 2016. – last visited on 2016/09/04

[26] PINHEIRO, Francisco A. C. ; GOGUEN, Joseph A.: An Object-Oriented Tool for Tracing Requirements. In: *IEEE Software* 13 (1996), S. 52–64

[27] RANGANATHAN, C. ; DHALIWAL, Jasbir S.: A survey of business process reengineering practices in Singapore. In: *Information & Management Vol. 39, pp. 125 to 134* (2001)

[28] RUSSO, Alessandro: *Data-aware and Artifact-centric Business Process Management*. Seminar Slides. http://www.dis.uniroma1.it/~arusso/didattica/s4i1314/Data_BPM.pdf. Version: 2013. – last visited on 2016/09/19

[29] SUN, James: *Barcelona Demo for ICSOC 2013*. video. https://www.youtube.com/watch?v=adw1rzB-64I&feature=youtu.be. Version: 2013. – last visited on 2016/09/19

[30] WEBSITE appium.io: *website appium.io*. Website. http://appium.io/. Version: 2016. – last visited on 2016/09/18

[31] WHITE, Stephen ; MIERS, Derek: *BPMN Modeling and Reference Guide*. Future Strategies Inc., Book Division, 2008

Name: Florian Rotter Matrikelnummer: 761110

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den . . . .

Florian Rotter