Faculty of Science, Technology and Communication

Visual Modelling of and on Tangible User Interfaces

Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of Master in Information and Computer Sciences

Author:
Eric Tobias

Supervisor:
Prof. Dr. Pierre Kelsen

Reviewer:
Prof. Dr. Yves Le Traon

Advisors:
Dr. Eric Ras
Public Research Centre Henri Tudor
Dr. Nuno Amalio

August 2012

Abstract

The purpose of this thesis is to investigate the use and benefit of visual modelling languages (VMLs) in modelling on tangible user interfaces (TUIs), and modelling TUI applications. Three main research questions are asked:

- Is it possible and practical to model an application or process using a TUI?
- Which General-purpose VML (GPVML) performs best in modelling a VML scenario for use on TUI?
- Is it realistic to use a GPVML to model complex TUI applications?

To answer the first research question, a Business Process Model and Notation, version 2 (BPMN2), ideation scenario is used with a tangible widget toolkit prototype on a TUI table to evaluate the performance and obtain feedback from test candidates. The study showed that it is possible to model a process using a TUI and the test candidate feedback did not lead to the conclusion that the modelling was cumbersome or impractical.

To find a suitable GPVML, the thesis explores and evaluates the current state of the art in VMLs for describing general problems. After gathering different VMLs, the thesis compares three languages using a simple scenario: the visual object constraint language (VOCL), augmented constraint diagrams (ACD), and the visual contract language (VCL). A weighted evaluation based on multiple quality criteria led to the conclusion that VCL is best suited to model TUI applications, answering the second research question.

The thesis answers the third research question by using VCL to model a more complex and complete scenario of an ideation process, which is based on using a BPMN2 on a TUI. This is done to assess VCL's suitability to more complex problems and its maturity. The study concludes that VCL is not yet mature enough to enable its general applicability in a wide variety of settings.

The three research questions were dressed with a hypothesis in mind: collaborative, novice friendly modelling environments are able to reduce the gap between stakeholders and software engineers during software projects, leading to a reduction of unrealistic expectations and an increase in the availability of domain knowledge. While the hypothesis is too broad to be proven by this thesis, the research questions answered here give some insights into how to approach it.

Acknowledgements

I'd like to thank the KISS team for having me and actively encouraging me. Special thank to Eric Ras, Olivier Zephir, Yves Rangoni, and Valérie Maquil. Many thanks also go to the LASSY, especially Nuno Amálio for helping me with VCL and ceaselessly working with Christian Glodt to add more functionality and fix problems with VCB. Thanks are also in order for Pierre Kelsen for his support and advice which led me to start my Master studies.

Sarah, thank you for encouraging me, being there when I need you, and making everything worthwhile! Misha, Susi, and Hues; thank you for brightening my days!

To Georges; may you find Irène waiting for you! I will never forget!

Contents

Chapter 1

Introduction

The rapid evolution in hardware meant that it was no longer a restraining factor for software. Inevitably, this lead, over the years, to a substantial increase in software complexity. In [1, 2], Brooks paints a gloomy picture of Software Engineering due to its incapacity to tackle the problem of rising complexity. The article, however, also stipulates that certain advances could be made during the next decades that would tackle the problem. In [3], Harel endorses Brook's point that there is no single simple solution to the software complexity problem, disagreeing, however, with his forlorn view of the field and advocating modelling as a means to tackle the complexity problem.

Harel [4, 3] argues that for software modelling a visual formalism is essential in helping to cope with software complexity. He states that; *"[...] the quality and expedition of [engineer's and programmer's] very thinking was found to be improved"*. However, Harel says that not enough data for any concrete statistical evaluation existed yet. Nevertheless, his observation is backed by research in the domain of Problem Solving in Psychology. Newell and Simon [5] found that the modelling of the problem space, as well as the extraction of concepts, is relevant to the problem, are important factors for the human problem solving behaviour. These observations can be applied to the domain of Software Engineering as well [6]. It seems that humans are naturally adept at using modelling and abstraction to solve problems.

Today, modelling has shown that it is suited to cope with the complexity of many different domains. For example, in Physics where the Standard Model [7] explains the undoubtedly intricate nuclear interactions affecting the dynamics of subatomic particles. Similarly, models that explain inherently complex interactions can be found, for example, in Biology [8] or Meteorology [9]. However, dealing with the complexity is not the only benefit of those models. They can be used to discover hidden behaviour, for example by model execution [3] or simply by the absence of an explanation

of an observed phenomenon [10], which was the case in the Standard Model where observations on broken symmetries and mass led to the prediction of the Higgs Boson [11].

Although there have been many improvements since Brooks' article, the software industry is still plagued by complexity related problems. Considerable amounts of money are lost every year due to the mismanagement of software projects according to the CHAOS report [12]. The major causes, according to that study are the lack of complete requirements, resources, user involvement and unrealistic expectations which made up for 46% of impairments during a software projects in 1995. It is not unreasonable to think that lack of user involvement leads to lack of domain knowledge, which in turn is likely to lead to incomplete requirements. In turn, lack of expertise in the software domain from the side of the stakeholder might lead to unrealistic expectations. The gap between domains and the lack of domain knowledge [13, 14] or *Domain-Expert Users* [15], seems to be one of the culprits for the failures in software projects.

This thesis investigates modelling in a collaborative environment. The idea is that a collaborative environment might help to close the gap by putting stakeholders and engineers working in the same environment. Due to the difference in domain knowledge however, stakeholders often lack the knowledge to understand models as is. Therefore, visual modelling languages (VML) will be used in the modelling process. A diagrammatic notation and the visual information contained therein can be better processed and recalled [16, 17] and is well suited as a medium to promote understanding of the model [18].

Understanding the model, however, might not be enough to investigate true collaborative scenarios. Furthering the ease of modelling itself and promoting the use of "natural" interfaces may further improve the acceptance and usability of the collaborative approach. Tangible User Interfaces (TUI) [19] use metaphors to naturally imply functionality and stimulate responses from users. For example, a tangible widget used to attach and move objects on the screen can simply be physically set onto the desired object and then moved. The combination of drop-move actions is a metaphor for reaching, grasping, and moving the objects. The use of metaphors is important [20, 21] as it eases the use of the interface, making it easy for users to anticipate functionality and understand the modelling concept. This eases working in a collaborative environment where multiple agents can communicate in natural language and realise their ideas seamlessly using a TUI.

The idea is that the joint modelling endeavour may remedy some shortcomings leading to project failures, such as unrealistic expectations [22] or

the lack of domain knowledge, by imposing that both parties are tightly involved in the process which, through the use of VML and TUI, is usable by engineers and stakeholders alike. The next section refines this idea into a set of concrete research questions. The remainder of the thesis;

- sheds light on the state of the art in modelling, VML and TUIs in Chapter 2,
- conducts a study existing VML and a comparison of their usability in Chapter 3,
- introduces the Visual Contract language in Chapter 4,
- investigates the maturity and usability of VCL regarding the modelling of complex TUI applications in Chapter 5,
- ushers a small usability study on the use of TUI driven modelling approaches in Chapter 6,

1.1 Research questions

The research questions address some aspects of the idea dressed in the previous section. They fall into two categories. One is to investigate the use of TUI and their suitability for the modelling process. The other category investigates the use of general-purpose VMLs (GPVMLs) to model complex applications.

- **RQ1** *Is it possible and practical to model an application or process using a TUI?*
- **RQ2** *Which General-purpose VML (GPVML) performs best in modelling a VML scenario for use on TUI?*
- **RQ3** *Is it realistic to use a GPVML to model complex TUI applications?*

Unfortunately, both topics are tightly intertwined and evaluating one after the other is, from a time perspective, not feasible. Therefore, the arrangement of chapters has been chosen to ease the flow of information and reading and does not respect the order of the research questions. The thesis investigates suitability and feasibility of TUI assisted modelling by building a TUI for Business Process Modelling Notation version 2 (BPMN2) [23]. BPMN2 uses a high level of abstraction that suits most business needs and is therefore suited to serve as a proxy given the scenario.

The first research question, RQ1, comes in two shades. First, it assesses the possibility of modelling using a TUI; this can be answered with a simple

yes or no upon formulating a strategy and making an attempt. Measuring the practicality is harder. Chapter 6 proposes a small example scenario using BPMN2 and a small sets of TUI components. The behaviour of the test subjects and their feedback will serve as a basis to answer all aspects of RQ1.

The second research question, RQ2 is tackled by designing and taking a small study which is detailed in Chapter 3. The performance will be graded and weighted to gauge performance. While the scope of the study is too small to be representative and total subjectivity cannot be guaranteed, the study allows for an interesting glimpse into the current major GPVML players and their abilities.

The third research question, RQ3, is rather large in scope. Chapter 5 elaborates on the modelling process using the language emerging as best suited from the study in Chapter 3. The complex application to be used as a basis for the case study is the scenario modelled to answer RQ1. To increase the complexity of the rather simple ideation scenario and thereby introduce an overall more realistic scenario, the modelling process will try to cover several modelling layers of the application. Not only will models of all involved domains be drafted but also their metamodels. The research question will be answered by posing several intermediary questions.

1.2 Research objectives

- Research the state of the art in VML, TUI and BPMN2.
- Design and conduct a case study to evaluate VML given a BPMN2 scenario on TUI to answer RQ2.
- Design and model a complex GPVML scenario for use on TUI to answer RQ3.
- Draft, conduct and evaluate a TUI modelling case study to determine the answer to RQ1.

Chapter 2

State of the Art

The following sections survey the literature in the domains of modelling, visual modelling languages (VML) and tangible user interfaces (TUI). Section 2.1 gives a brief overview of the modelling activity in Software Engineering and detail specific angles that will be leveraged in this thesis. Section 2.2 highlights the most notable VML of the recent years as discovered during the course of the study that will be detailed in Chapter 3. The section goes into details about some of the terminology and domain specificities of interest. Section 2.3 introduces the reader to TUI, including concepts used in these interfaces and a brief history.

2.1 Modelling

Humans are often confronted with difficult and complex problems that they are unable to solve on the fly. The most prominent approach in the domain of Problem Solving in Psychology has been formulated by Newell and Simon [5]. In their work, the modelling of the problem space, as well as the extraction of concepts relevant to the problem are important factors of the human problem solving behaviour which can be applied to software as well [6]. Hence, humans seem naturally adept at using modelling and abstraction to solve problems.

It is often through abstraction that models are articulated to be comprehensible and suited to the problem space. Today, modelling has shown that it is suited to cope with the complexity of many different domains. The problem of using abstractions is that they are usually specific to the author of the model and subjective. The meaning of these abstractions needs to be communicated properly to any reader [24].

An interesting aspect of modelling is that there is, dependent on the subject, more than one valid model. It is also possible to model a problem

from different viewpoints. It is paramount that viewpoints and abstractions be unambiguously communicated to the reader [24]. To achieve this goal, the use of some sort of formalism seems unavoidable. The use of formalisms has other benefits, such as, the possibility to use formal reasoning to prove their validity or at least those of their core concepts [25, 26].

The need for certification and assurance in the safety-critical systems industry, such as, Public Transportation [27, 28]. Formal models offer an elegant and, if applied correctly, efficient way of establishing the required trust by using techniques such as model checking [29, 26, 30]. It is however necessary to plan ahead and gather evidence and collect proof throughout the life-cycle of the software project which is not a trivial task [30]. Applying these techniques using established methodologies such as Model-driven Software Engineering for example is therefore recommendable.

Model-Driven Engineering (MDE) is an approach to Software Engineering that focuses on the use of models to abstract complexity. In contrast to Computer-Aided Software Engineering (CASE) [31], MDE does not fail to cope with platform complexity and does not try to implement a "one-size-fits-all" solution [32]. In [32], Schmidt refers to MDE as; *"A promising approach to address platform complexity—and the inability of third-generation languages to alleviate this complexity and express domain concepts effectively—is to develop Model-Driven Engineering (MDE) technologies that combine [domain-specific modelling languages, transformation engines and generators]"*.

Some noteworthy approaches implementing the MDE methodology include the Eclipse Modelling Project [33] and the Model-Driven Architecture (MDA) by the Object Management Group (OMG) [34, 35]. The latter proposes the use of platform independent models (PIMs) to define high level models of a system's functionality, stable with regards to requirements, and unaffected by technological changes as they simply do not include technical details. The idea is to transform or map from PIM to platform specific models (PSMs) if desired, to implement the application on a given platform [34, 36]. This scheme aims at solving some of the integration problems due to the number of different platforms that have emerged on the market.

To enable the transformation from PIMs to PSMs, MDA requires the use of meta object facility (MOF)-based languages when specifying the PIM [34]. The MOF specification [37] also documents a four layer meta-modelling framework. These layers, as shown in Figure 2.1 range from *m0* to *m3*. MOF is situated at the *m3* level. The bottom most layer, *m0*, contains the model instance of the modelled phenomena. The *m1* layer provides a model of the phenomenon, a means to express it in a more abstract fashion.

Abstracting this model, *m2* models the *m1* model itself, providing a mean to its instance to use different language constructs to delimit objects, express properties, relations, etc... On the topmost layer, in the case of MDA, MOF, the model defines itself recursively as well as defining what model elements are and how they relate. An example can be found in Figure 2.2.

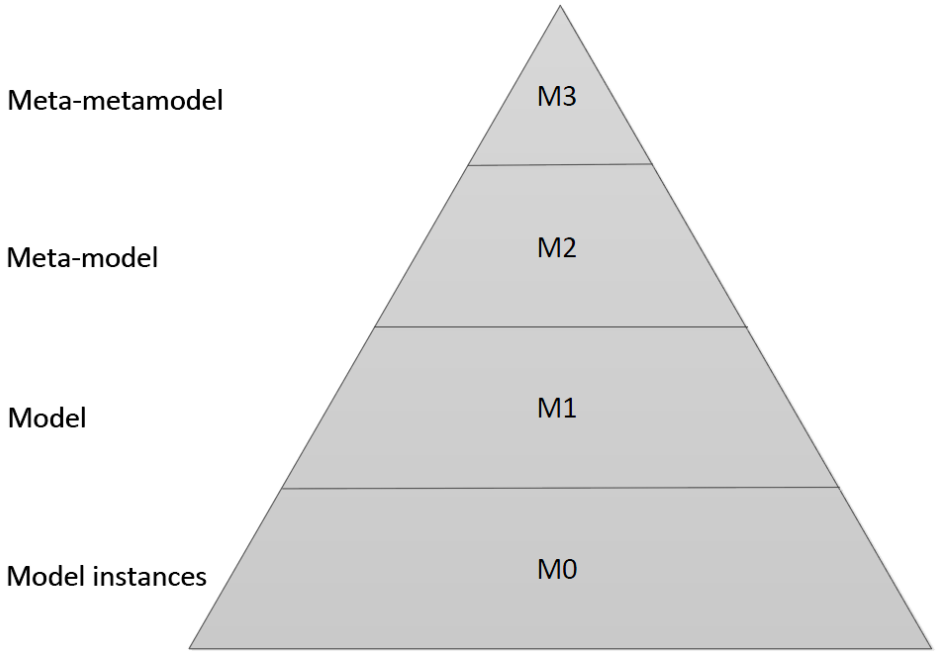


Figure 2.1: The different modelling layers after [37]

While this model specifies different layers of abstraction, it is not a trivial task to actually find the right level of abstraction. This process usually boils down to a trial-and-error process which is unacceptable for an engineering field [24]. Also, Kent mentions in [36] that there is a horizontal layering that is untapped by the OMG. He also mentions that several initiatives have begun looking into the mater such as for example Aspect Oriented Software Development [38].

An interesting aspect of MDA is that is stipulates that model translations can be automated [35]. Kent implies that MDA, while not insisting on the matter, intents to automate the mapping between PIM and PSM [36]. However, this automated is not limited to higher level of abstractions, automatically generating code from a PSM is a possibility [3, 32, 39]. However, the result of these translations can also be simulated instances, sample states, or scenarios. On these, analysis can be performed to gain insights into the correctness of the parent model [26, 3].

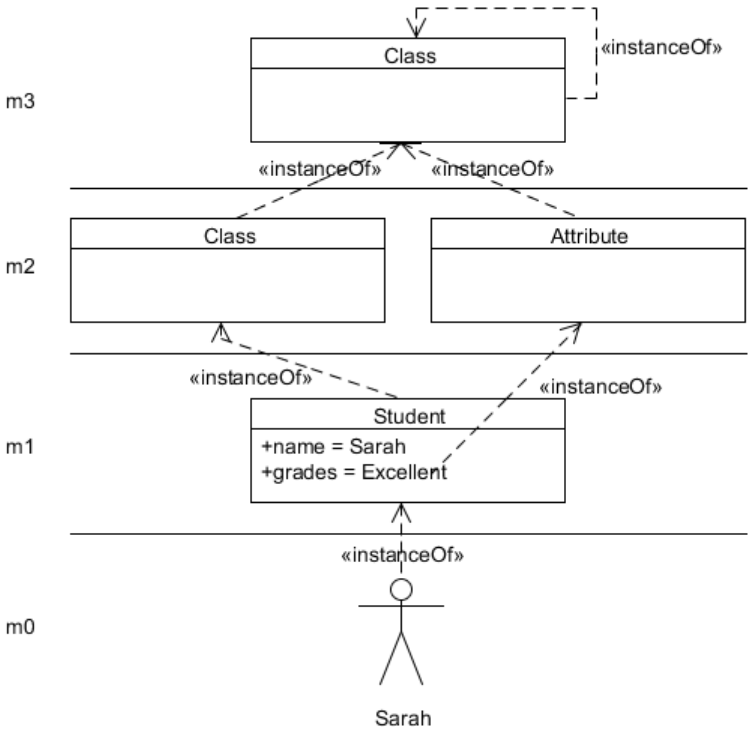


Figure 2.2: A concrete example illustrating Figure 2.1

2.1.1 Definitions

When talking about different modelling layers, it is important to be on the same page terminology-wise. Throughout the literature, the term "model" is well understood. An excerpt from the Merriam-Webster online dictionary [40] defines model as;

- "*a description or analogy used to help visualize something (as an atom) that cannot be directly observed*"

- "*a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs; also : a computer simulation based on such a system <climate models>*"

Alas, there is among experts no consensus on a definition of "metamodel" [41]. Therefore, in this thesis, a metamodel shall simply be considered the description of a language to build models.

2.2 Visual Modelling Languages

Languages serve as a tool in communication. Natural languages are what people use on a daily basis. They are the medium that transports different kinds of information from the author or speaker to the reader or listener. However, two speakers might not choose the same wording or symbolism to convey the same meaning. And even more frightening from an engineering viewpoint is that the same message and bits of information can be interpreted differently by multiple receivers, each tainting the message by his personality and experiences. Natural language is ambiguous [42].

Natural language does not provide the necessary formalism to be useful in this context. Artificial, formally defined languages, usually do not suffer from these shortcomings. They are important for conveying unambiguous information [43]. According to Slonneger and Kurtz, they are build from the same blocks as natural languages, syntax, semantics and pragmatics. The syntax defines the grammar of a language, what arrangement of symbols is correct. To infuse meaning into the structure of symbols, semantics imprint these structures with a concept. Pragmatics transcend the language itself and deals with language user perception and subjective feeling such as ease of use or efficiency.

Modelling languages are such artificial languages. Not all modelling languages are however defined formally in full [44, 45]. While it may not be easy to reach a complete formal definition, it is a myth that a formal definition is harder to read for stakeholders and can therefore be neglected. In [46] states that a formal specifications can be paraphrased or wrapped to present a more palatable document to the user but that they do help users understand. Indeed, if an unambiguous understanding of the specification can be achieved, unrealistic expectations are less likely to emerge.

Modelling languages can be split into two categories. Domain-specific languages (DSL) [47] and general-purpose languages (GPL).[1] DSL are tailored to a specific domain to solve a given set or a single problem. Their scope is as narrow as the domain and their semantics, syntax and use of symbols adapted to suit modelling the domain. Examples of DSLs are;

- MOF [37], a language to build metamodels,
- VHDL [48], a hardware description language for describing integrated circuitry.

[1] In this context we are talking about modelling languages. Therefore the abbreviations DSL and DSML respectively GPL and GPML are used as synonyms.

GPL are not tailored to a particular domain and specify a domain neutral syntax and semantics. Some examples of GPLs are;

- UML [49], a modelling language for modelling software, especially object-oriented, systems,
- VCL [50], a modelling language based on set theory for modelling software systems,

Discussions in the field seem to favour the use of DSL as their specialisation is optimised to address specific aspects [26]. In [51] an empirical study suggests that DSLs are superior to GPLs in all cognitive dimensions. This thesis will not take any sides as it would require a substantial effort to make a concise observation. Due to the objective of the thesis and the research questions dressed in Section 1.1, only GPLs are considered due to them being able to address problems in multiple domains.

In Section 2.1, the argument was made that it was natural for humans to abstract and model complex problems. Visual notations improve the ability to understand these models [4, 3, 18] as our physiology and cognitive process is apt at dealing and processing visual information [18]. Visual modelling languages (VMLs) use a diagrammatic notation to capitalise on this human ability. Their strength lies in combining purely diagrammatic information with topological or spatial information [4, 18]. Nevertheless, VMLs should not sacrifice formalism for the sake of visual expressiveness.

VMLs are built from the same building blocks as regular modelling languages. They feature a syntax and semantics. Moody argues in [18] that the benefits of the visual nature of VML does not come automatically, it must be designed into the language. He further states that semantics are valued more than syntax when evaluating VML. That there is a need for formalism and hence a coherent and correct syntax has been argued in section 2.1. However, it is a misconception that visual languages are not formal [52]. This is likely linked to the myth that the use of formalism require advanced mathematical skills [46].

2.2.1 VML's many influences

Some of the earliest topo-visual formalisms, according to Harel [3], were developed by Euler and Venn in the 18^{th} and 19^{th} century respectively. With the contribution of mathematicians such as Cantor or Dedekind, these early concepts were formulated into what we know today as Set Theory. The latter influenced many novel approaches in modelling such as Higraphs [3], Constraint Diagrams (CD) [53], or Visual Contract Language (VCL) [50]. The

Entity-Relationship Model [54] together with Flow Charts, a generalised form of Flow Process Charts proposed by Frank Gilberth, found in [55], stand, amongst others, as inspiration to the Unified Modelling Language (UML) [49] [2].

All of these languages share the need to express constraints on their models. While it is best to craft the ability to express constraints into the language by using its topo-visual nature [18], UML for example uses the Object Constraint Language (OCL) [56] to declare constraints in a notation akin to mathematical logics. First Order Logic (FOL) does influence the expression of constraints and invariants in many VML such as, for example, UML and VCL. Other examples of VML can be found in Chapter 3.

2.3 Tangible User Interfaces

In [57], Shneiderman observes that interfaces seldom allow for immediate visualisation and manipulation of data and are therefore not suited to represent the fast moving world where information is ever changing and manipulations need to be precise, simple and reversible. Furthermore, they do not tap into the capabilities of the human visual cortex when they display only a handful of items and interface objects. He believes that interfaces should encourage the user to explore all possibilities by trial and error but that interfaces are seldom capable to predict and hence deal with all possibilities. In [58] goes one step further. He deplores the inability of current UI to draw upon the human senses such as touch and propose manipulations more natural to humans and the domain they are viewing.

Weiser's vision of Ubiquitous Computing [59], of seamlessly integrating the digital into the physical, making it go unnoticed is one of the core principles of Tangible User Interfaces (TUIs) [19]. They blend digital information into a physical form [60], allowing the manipulation of the digital through natural, haptic interaction with the object which, by its construction and make, serves as a gateway to the information linked to it. Hence, users can use all their senses and skills to interact with TUIs without being limited by technical constraints.

For the TUI to be usable in a such a natural fashion, using not only the user's visual capacities as traditional GUI do, but to also convey information on a haptic and topological level, the interface must apply metaphors to imply functionality [21, 20]. The application of metaphors works by designing the physical part of the interface, the tangible widgets, to imply a

[2] Currently at version 2.4.1 `http://www.omg.org/spec/UML/2.4.1/` Accessed 13 April 2012

metaphorical context [61]. The implication of functionality relies mainly on the shape, colour and texture. However, any physical stimulus can be used to imply functionality such as smells or sounds. The metaphorical context allows for the application of well known metaphors to be used, mapping the source domain containing a familiar concept such as rotating a radial or shaping clay, into the target domain altering the state of the digital. This is called a metaphorical projection.

The meaning expressed by the metaphorical projection resided in the application domain. The metaphor helps the user understand the meaning in the application domain through the use of familiar, borrowed concepts from the source domain in the unfamiliar, digital target domain. [20, 62]. An example would be Photohelix [63] [3]. The rotative tangible widget or handle together with the visualisation of the pictures arranged in a helix imply that rotating the handle will result in a rotation of the helix. Users will immediately understand the metaphor and all its implications such as the rotation's direction and its effect.

One of the first implementations of TUI to draw public attention was Durell Bishop's Marbel Answering Machine in 1992, described in [64]. It encoded incoming calls onto marbles which could then be replayed in any order, removed from the users to use in any way they seemed logical, like labelling them or placing them in storage, or used with other devices to retrieve stored informations such as phone numbers. Since 1992, many TUI have been implemented. Recently, developments, notably those from the MIT Media Laboratory [4], and its Tangible Media Group [5], both of which Ishii is involved with, have produced several prototypes. For example Urp [65, 58] which is a system for urban planning or Illunimating Clay [66, 58], a system for real-time computational analysis of landscape models [6].

A lot of the recent research in tangible media has been developed with table tops in mind. A milestone which has triggered a lot of research and implementations using tabletop TUIs has been reacTable [67], implementing an electronic music instrument. Users set tangible widgets on the surface to produce sounds. They can then alter the sound's frequency and amplitude as well as introduce control elements, thereby altering the audio [7]. The

[3] See `http://www.youtube.com/watch?v=ReO36uSr_SY` for a video demonstration. Accessed 30 July 2012

[4] MIT Media Lab `www.media.mit.edu` Accessed July 27 2012

[5] MIT Media Lab – Tangible Media Group `http://tangible.media.mit.edu/` Accessed July 27 2012

[6] To see Illuminating Clay in action, visit `http://zomobo.net/mit-media-lab-tangible-media-group-illuminating-clay` Accessed 26 July 2012

[7] Visit`http://www.youtube.com/watch?v=PGiasLiGTX4` for an example of reacTable

work spent on the reacTable was refined and distilled into the reacTIVision framework [68]. The framework provides facilities for tracking on table objects via fiducial markers attached to their base via a below-table infra red camera. The table at Public Research Centre Henri Tudor which this document uses as a reference for investigating TUI interactions as well as designing TUI scenarios is based on reacTIVision.

Figure 2.3: The reacTable. Picture by Daniel Williams.

Tabletop TUIs have a desirable property. They, due to their size and placement, facilitate multi-user participation, enabling this type of TUI to be used in scenarios where collaboration is beneficial [69]. A collaborative environment might positively affect the outcome of a modelling process if stakeholders respectively domain experts and engineers work in the same environment. The effects could be to reduce unrealistic expectations [22] and improve the availability of domain knowledge. Both factors are linked to failures in software projects [13, 14, 15].

This thesis will not be able to fully investigate the matter but be content itself with investigating the feasibility of using TUI to model software applications or business processes and to identify suited modelling languages as given by the research questions in Section 1.1. All of the shortcoming regarding the overall hypothesis will be covered in Section 7.1, Future Work.

Chapter 3

Case Study

This study is designed to answer the second Research Question found in Section 1.1;

RQ2 *Which GPVML performs best in modelling a VML scenario for use on TUI?*

The study is composed of four objectives. The first objective is to scavenge the domain of Visual Modelling Languages (VML) in order to identify all VML that satisfy several requirements. These requirements will then be refined to obtain a small set of mature general-purpose VML (GPVML). Section 3.1 will describe all steps and actions taken. Section 3.4 will describe the design of a small scenario which is modelled in Section 3.5 by the pre-selected set of VML. The results are evaluated by the measurements detailed in Section 3.6 and presented in Section 3.7 before resuming and concluding the findings in Section 3.8. The study has also been compiled into a technical report which contains an exhaustive documentation of all resources and methodologies [70].

3.1 Identifying suitable languages

Before beginning with studying suitable languages, a preliminary set of languages is collected. The criteria dressed here only serve to limit the number of languages taken into considerations. With the sheer number of languages and dialects produced by the scientific community in the domain of Software Engineering it would be impossible to undertake an exhaustive comparative study in the scope of a master's thesis. These criteria are by no means globally classifying languages by quality nor should the criteria be interpreted as a sign of suitability other than their relevance regarding the work in this thesis. The following sections detail the three criteria that were used to select languages. Only those meeting these three criteria were retained.

Visual Syntax The main delimiting criteria is the requirement on languages to have a visual syntax. This strong mandatory criteria is inherited directly from the scope of the thesis. One of the reasons behind selecting and investigating only visual languages is the tease of using Tangible User Interfaces (TUIs) to dress the models for future TUI applications. Languages should be visual in order to benefit most from TUI by enabling the use of and thereby benefiting from the power of metaphors. Non-visual languages would either need to be translated which is a rather costly process or would not benefit from using TUI.

One of the strong points of TUI is the ability to work cooperatively [69]. In order to benefit from collaboration in a business modelling scenario, as detailed in Section 3.4, bringing the technically non-versed customer together with business analysts and experts is important [13, 14]. Using a VML for the collaborative scenario will ease the collaboration as metaphors and visuals are easier to process [16, 17] for non-expert users. Staying close to and integrating the customer into the design process is desired as it is thought to prevent problems during product validation and increase the quality of the final product [13, 14, 15, 22]. It is one of the reasons why recent Software Engineering and Project Management methodologies such as for example Agile, have integrated ongoing customer feedback into their core teachings [71].

Modelling language Another criteria directly inherited from the thesis description is the need for the language under investigation to be a modelling language and not a programming language. Should any language fall into both categories it will be retained. Other than not fitting inside the scope of the thesis, it would make little sense to compare languages with different application domains.

General-purpose language The scenario used to answer the research questions, drafted in Section 1.1 RQ1 respectively RQ2, is based on the Business Process Modelling Notation version 2 (BPMN2) [23]. To answer the remaining research question, RQ3, models and metamodels for BPMN2, the TUI and the application, mapping TUI onto BPMN2, will be drafted. Since there is not one specific domain, a domains specific language (DSL) would be unsuited for the modelling process, hence, one reason to choose a general-purpose modelling language (GPML). Moreover, while this study only uses a business scenario based on a formally defined specification, future modelling processes might take a more liberal approach and need a more flexible, general-purpose modelling language.

Choosing a GPML also offers the benefit of not being overly specialised and catered to one particular application domain. This enables the language to use well known metaphors and abstractions to draw upon existing language schemata and improve language learning process [21]. As noted by Schema Theory [72], the ability to draw upon existing schemata will improve the ability to understand the new domain by reusing familiar concepts in already existing schemata. The more specialised and tailored metaphors that would need to be used with DSL might not have that same benefit considering the mix of expert and non-expert users.

3.1.1 Collection

After a review of existing literature using common search engines and skimming references, more than two dozen relevant resources were gathered. For a full list, please refer to the Appendix F. These resources can be classified into one of the four categories;

- papers and other resources describing modelling or simulation environments such as for example AnyLogic [73] or VENSIM [74],
- studies on visual notations, their application and use, as well as on emerging visual paradigms. A few examples of such resources are *"Analysing the Cognitive Effectiveness of the UCM Visual Notation"* [75] or *"Towards Symbolic Analysis of Visual Modeling Languages"* [76],
- extensions or enhancements to existing visual notations such as for example *"The semantics of augmented constraint diagrams"* [77],
- papers and other sources describing novel visual languages such as for example UML [49] or Constraint Diagrams [78].

3.2 Pre-selection

In order to be able to make a final choice, the number of candidate languages had to be reduced. The following paragraphs explain what resources and thereby languages were discarded and for what reasons. Please note that the order was chosen by what subjectively seemed to eliminate the most resources. The elimination criteria were those elaborated above.

The first few resources to be discarded were all detailing simulation and modelling environments that were not providing any syntax outside of their environment. The affected items were related to AnyLogic [73], VENSIM [74], SIMILE [79] and subTextile [80]. The latter offers a very

interesting approach for designing interactive systems. This could probably be used for designing widgets but the specificity of the visual programming language and hardware platform means that subTextile will not enable it to model many different domains. Moreover, it focuses strongly on behaviour, neglecting structural elements. VENSIM is a simulator that allows modelling and simulating business, scientific, environmental, and social systems with a focus on system dynamics and interactions. AnyLogic is more powerful but covers the same niches. Both simulators allow to model complex behaviour but are limited to the domain of system dynamics. By modelling and thereby preparing the system to base simulations on, the focus of the semantics lie more on defining agents, behaviour and interactions. Each simulator has a pragmatic approach to define the semantics which are not very formal or usable without the tool.

The unnamed language that was presented in [81] is designed to support sketch based design in the field of interface design. While the language could be ported to similar disciplines such as storyboarding or design in general, the applicability of the language to a broader field is questionable. The language was deemed to be domain specific and was therefore discarded. PROGRESS [82] is a visual programming language using graph rewriting systems. It is however not a modelling language and has therefore been removed from the set of considered languages. The same reason holds for SPARCL [83], a visual logic programming language based on set partitioning constraints, and Forms/3 [84], a first-order visual language to explore the boundaries of the spreadsheet paradigm. The approach taken in [85] describes a novel visual programming language to draw and execute flowcharts aims at shifting the focus from code generation to algorithmic conception of programs. Not providing a general-purpose modelling approach, it was dropped from the set of considered languages. Due to the domain specificity of [86] it did not fall into the scope of this work. The Regatta approach as described in [87] is similar in semantics to BPMN2. While the approach could be broadened to include more domains, the specific aim of the approach is Business Process Engineering.

A last step of the preliminary sorting before tightening the criteria is to remove all stepping-stone-papers that had been used to uncover more languages but describe by themselves no language or language extension; eliminating [75, 76, 88, 89, 90].

3.3 VML selection

For a VML to be selected it;

- must be able to model the abstract, high level view of the customer, refine the abstract model into ever more elaborate processes until the complete system had been modelled, modelling all structural and behavioural requirements,
- must be able to express constraints on those structures and behaviours,
- must allow to express associations and dependencies between structural and behavioural elements,
- must express the system's behaviour regarding inputs and outputs.

Ideally, all of these requirements and constraints would need to be represented visually as a formal textual representation might prove a significant entry barrier for novel users.

The paper *"Visual modeling of OWL DL ontologies using UML"* [91] introduces a visual, UML-based notation for representing OWL ontologies. Ontologies inherently only express domain concepts and how they are related. The same limitation is faced by Object-Role Modelling [92] which focuses on structural concepts and their relation. Hence, behaviour and complicated constraints are not taken into consideration by these notations. A paper on visual constraint programming [93] has similar issues. The focus lies on constraints and the other requirements are neglected. Moreover, this paper takes on a programming rather than a modelling approach. The remaining thirteen papers and resources can neatly be arranged as specifying, describing, or studying one of the following three languages;

- Unified Modelling Language & extensions [49, 94, 95, 96],
- Visual Contract Language [50, 97, 98, 99],
- Constraint Diagrams & extensions [53, 78, 100, 101].

3.3.1 UML & VOCL

The amount of material found regarding UML is not surprising as it is a well established standard which has a vast user base in the software industry. The language is generic enough for the goals of this study while still offering all the tools needed to model the structure and behaviour in the domain. However, in order to satisfy requirements on correctness and quality, constraints are needed. This is not possible in pure UML and requires the use of an extension. Two extensions that seemed to fit the job description at

first were the Object Constraint Language, OCL [56] and Executable Visual Contracts (EVCs) [102]. However, EVCs seemed more limited in expressiveness, hence, the use of OCL was preferable. Unfortunately, OCL does not offer a visual representation and is thought to be cumbersome [78]. To that end a Visualisation of OCL or simply Visual OCL (VOCL) [94, 95, 96] is used.

VOCL extends the formal declarative text language OCL with a visual syntax akin to that used in UML. The objective of the visualisation is to further the use of formalisms to express constraints as [94] hints that the lack of said use is due to its mathematical and formal nature. Using UML to express structural and behavioural requirements while using VOCL to express quality requirements and invariants is suited for modelling the scenario.

3.3.2 VCL

The Visual Constraint Language (VCL) [50, 97, 98, 99] is a recently developed language building on the Set Theory. The language is inspired by Higraphs [4] and UML class diagrams, the latter of which was influenced by Entity-Relationship diagrams. VCL takes a step further in that it adds assertions into the structure to increase expressiveness. VCL provides a mean to not only separate concerns using modular design but also to address the different requirements separately, much like UML. Structural requirements are modelled separately from behavioural requirements which enables modellers to focus on one concern at a time. Instead of using multiple diagram types to specify behaviour as applied in UML, VCL subscribes to the Design by Contract paradigm [103] and keeps the expression of pre- and postconditions syntactically similar to the other diagram types. Global and local correctness requirements are addressed by so called assertion diagrams which express invariants.

As the language was developed recently by University of Luxembourg, research and improvements in and to the language are still ongoing. The version of VCL and the accompanying tool have changed during the thesis which leads to some improvements in later models in regard to expressiveness and visual syntax. Chapter 4 will provide a more in depth look at VCL.

3.3.3 Constraint Diagrams

Transmitting knowledge through diagrammatic notation is an important step in making a domain accessible by outsiders. To this goal, visual languages and notations have been developed for quite a while. Some pioneers like Euler (18^{th} century) and Venn (19^{th} century) recognised the need for visualisation and introduced diagrammatic notations for mathematical and

logical constructs. Constraint Diagrams (CDs) [100, 101] is a notation that is very close to those early notations and provides a diagrammatic notation for expressing constraints similar to first order predicate logic [53].

For the purpose of this study, the plain CDs as proposed by [78] are not sufficient. The initial purpose of CDs was to serve as an alternative OCL. However, an augmented form of CDs (ACDs), proposed in [53], extends the language and allows the specification of structural requirements. Behaviour is proposed to be expressed in a Design by Contract [103] like precondition-postcondition manner.

3.4 Designing a scenario

In Section 1.1, when dressing the research questions, it was made clear that it would be impossible due to time constraints to investigate one matter after the other. Therefore, the Business Process Modeling Notation version 2 (BPMN2) [23] was chosen to model a scenario. BPMN2 while a domain specific language, is not as domain specific as one might think. Its design close to flow charts is apparent and as the notation leaves a lot of freedom to the designer, it can be used to model very abstract business problems.

The scenario modelled here takes on a simple business process, managing a stock portfolio. The BPMN2 model is explained in detail in Section 3.4.1. Furthermore, a simple TUI application, Section 3.4.2, is conceived in collaboration with Paul Bicheler. This preliminary TUI model is used to extract a small number of relevant tangible widgets in Section 3.4.3.

3.4.1 BPMN2 introduction and model

The BPMN was developed by the Object Management Group (OMG) and is currently in its second version [23]. The specification introduces the notation perfectly for the context in which it will be used;

> "*The primary goal of **BPMN** is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, **BPMN** creates a standardized bridge for the gap between the business process design and process implementation [23].*"

The BPMN2 scenario shown in Figure 3.1 is a simple, stock themed, fictive, client-service provider model. The model is an instance of the BPMN2

metamodel found in [23].The focus of this exercise is not to model all aspects of the scenario as accurately and diligently as possible as a lot of time would need to be spent on that endeavour. Most important is the modelling of the perceived visual model without worrying to much about the underlying metamodel. A metamodel conform BPMN2 model will be drafted later in Chapter 5. Said chapter will also include a more technical explanation of the notation and notes in its usage in this document.

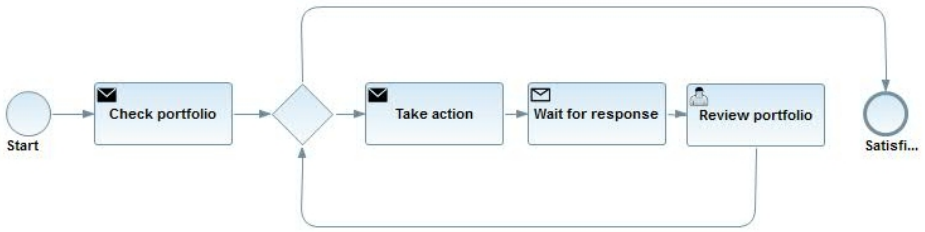


Figure 3.1: BPMN2 stock management scenario

Figure 3.1 shows the scenario which only holds one process from the customer's view. The scenario sees the customer check his portfolio and then either take action or end the scenario. After taking an action he will have to wait for a response before a final review of his portfolio. He can then choose to take another action or end the scenario. This simple introductory scenario covered a few basic aspects of BPMN2 and allowed to test some basic TUI-based modelling approaches. The scenario was refined three times to produce more finely grained scenarios. However, due to time constraints, only the first scenario was modelled, hence, the remaining scenarios are not relevant. For documentation purposes, the document detailing all scenarios, *"Capital Stock Management scenario description document"*, can be found at; `http://tinyurl.com/bqpqdj2`.

3.4.2 Preliminary TUI model

The preliminary TUI model was drafted in cooperation with Bicheler who is also following an internship at the Public Research Centre Henri Tudor. All widgets conceived with said model can be found in Tables 5.1 to 5.3 in Bicheler's thesis [104] or in Appendix J. At the time the study was designed, no coherent model or understanding on the building blocks of the TUI were available as it relied on the work in progress for Bicheler's Master's Thesis [104]. Therefore, all attempts to model TUI were based on efforts to formalise the notion of TUI and the related terminology such as widgets or zones.

In order to keep the modelling efforts comparable, a number of requirements were drafted in cooperation with Paul Bicheler. These requirements spanned structural and behavioural aspects and detailed a few simple constraints. For example, a tangible widget must have at least one handle or a handle may be bound. Such atomic requirement statements are of course far removed from what a customer might specify but the simple nature of the requirements was preferred as it meant that time was spent on the essential, modelling. The requirements have been compiled into a list and can be found in Appendix G.

3.4.3 A simple TUI instance

The preliminary TUI model was the basis for choosing a set of widgets used to model the BPMN2 instance. Considering the BPMN2 scenario to be modelled from Figure 3.1, only a subset of all the widgets contained in the preliminary TUI model were needed. Table 3.1 lists all widgets in use.

3.5 Case study

During the study, tree previously identified VMLs; UML & VOCL, VCL, and ACD were used to model the scenario defined in 3.4. Before each modelling attempt, a few days were spent on getting acquainted with the languages, modelling some tutorials if available and consulting some papers highlighting examples.

The following sections will explain which measurements were collected in order to be able to evaluate the chosen VMLs in Section 3.5.1. Section 3.5.2 will contain a rundown of the actual study. Section 3.5.3 wrapping up the study, showing and commenting the models that were produced.

3.5.1 Measurements

In order to be able to compare the different VMLs, measurements are needed that can be objectively collected and evaluated. The measurements are split into six categories. *Tool support* is aimed to measure the availability and quality of language accompanying tool. The use of a tool is usually preferable as it usually offers many facilities that make modelling easier and less error prone. *Semantics & Transformation* captures the degree of formalism the VML offers. The *Expressivity* category captures the number of requirements that have been met by each language and computes a ratio. The *Usability* of each VML is measured by the category with the same name. *Error Checking* puts a number to the facilities each VML and its accompanying tools offer for rooting errors during the modelling process. Finally *Verification* measures modularity and the scheme used to formally verify the

Name	Physical Action	Intent	Result
Stamp [Create component]	The actor moves the widget onto the Toolbox, selects a component by activating the widget and then stamps one or multiple components onto the canvas.	The actor wants to select a component from the Toolbox and place it once or multiple times onto the canvas.	The desired number of components is placed on the canvas.
Stamp [Link components]	The actor moves the widget onto the Toolbox, selects a component by activating the widget and then stamps one or multiple components onto the canvas.	The actor wants to link existing components with the selected component or place it inside existing components by placing the focus point onto an existing component.	The existing components are linked by the new component or the new component is placed inside the new component.
Chain	The actor activates the widget on a component, selects a connector component by activating the widget and then slides the widget to a destination and activates it to confirm and select the kind of endpoint component.	The actor wants to create a new element by creating a link and endpoint from an existing component.	A new connector component and linked non-connector endpoint component are created.

Table 3.1: Table of widgets used in the study

model. Table 3.2 lists all measurement categories and their sub-categories. For details on how each category was gauged, please consult Section 3.6.

3.5.2 Study walkthrough

In order to be able to compare the modelling performance of the languages given the scenario, a plan had to be drafted and followed. Therefore, it seemed unwise to simply take all requirements and model them sequentially. The following paragraphs will detail the step-wise modelling process that was followed using each modelling language. All requirements can be found in

Measurement categories	
Tool support	Availability Maintained Latest version Branch
Semantics & Transformation	Formally defined Transformability
Expressivity	# [X] [1] # [X] satisfied # requirements partially satisfied # unsatisfied requirements Ratio
Usability	Naming conventions Naming fit Documentation Tutorial Hands-on-tutorial Primitive mutability Live suggestions
Error Checking	Time Syntax highlighting Degree Error correction suggestion Debugging possible
Verification	Modularity Verification scheme

Table 3.2: Table of measurements

Appendix G. The requirements are denoted by an abbreviation that follows the naming convention; [*Domain*][*Kind*][*Running number*] where *Domain* is either *BPMN2 (B)*, *BPMN2 Stock Management scenario (S)*, or *TUI (T)*. The *Kind* denotes the nature of the requirement and can be either a *structural requirement (S)*, *behavioural requirement (B)*, or a *constraint (C)*. Therefore, **TB2** denotes the second behavioural requirement of the TUI.

Preliminaries To be able to execute the following eight steps of the scenario, complete BPMN2 and TUI models needed to be drafted. Hence, all the following requirements needed to be satisfied;

- **BS1** to **BS15**,

[1] Where X is either structural requirements, behavioural requirements or constraints.

- **BC1**,
- **TS1** to **TS3**,
- **TB1** to **TB4**,
- **TC1**.

Create Start Event To create a Start Event we need a widget in Stamp mode. We therefore must fulfil **TS4** and **TS6**. As for widget behaviour, the Stamp requires **TB5**, **TB16** [2] and **TB17**.

Create Activity To state all requirements needed to create an activity we need to cater to both creation scenarios. The use of the Stamp is covered by the previous task so we can simply assume we need all those requirements too; **TS4**, **TS6**, **TB5**, **TB16** and **TB17**. To use the Chain widget we also need **TS5** and more items from**TB16**. In addition we need to meet several requirements from the stock management model, depending on the type of activity that is created.

Check portfolio In order to attribute the activity to a user said user and his portfolio are needed; **SS1**, **SS5** and **SS6**. Moreover, the related behaviour and constraints; **SB1** and **SC4** are needed.

Take action Information about the customer, **SS1** and about an eventual order the customer may place, **SS4**, **SB2** and **SC1** is needed. To place an order, requirements catering to the subject of orders, stock, are needed too; **SS2** and **SS3**. Due to the encapsulation of information, details about the orders are not noted in BPMN2 and hence the differentiation between stocks for example would only become apparent if a decision is taken based on their difference.

Wait for response Information about the customer, **SS1** and the action he may take; **SB4** is needed.

Review portfolio In order to attribute the activity to a user said user and his portfolio are needed; **SS1**, **SS5** and **SS6**. Moreover, the related behaviour and constraints; **SB3** and **SC4** are needed.

Create Gateway / Create Event Analogous to Create Activity.

[2] For **TB16**, only those rows are required that list the mode in question.

Link components To create a connection we need a widget in either Stamp or Chain mode. The Stamp requires that **TS4** and **TS6** must be fulfilled. As for behaviour, the Stamp requires **TB5**, **TB16** and **TB17**. To use the Chain widget **TS5** and more items from **TB16** are needed.

3.5.3 Product

At the end of the modelling process, each language had produced multiple models which will be shown here. The subjective experience gained during the modelling process will be reflected during the evaluation phase detailed in Section 3.6. The following paragraphs will comment on the modelling process as well as include the models that were produced.

3.5.3.1 UML & VOCL

VOCL serves as a visualisation for OCL hence the reason why structure and behaviour of all related domains are modelled in UML using the stand-alone version of UMLet [3] downloadable at `http://www.umlet.com/`. Figure 3.3 shows the final result of this model after the entirety of the scenario had been modelled. The constraints for this level are expressed in VOCL using an Eclipse VOCL-Plugin from the TU-Berlin. The plugin was developed during a student project and can be downloaded at `http://tfs.cs.tu-berlin.de/vocl/userguide/group1/`. The constraints can be seen in Figure 3.2.

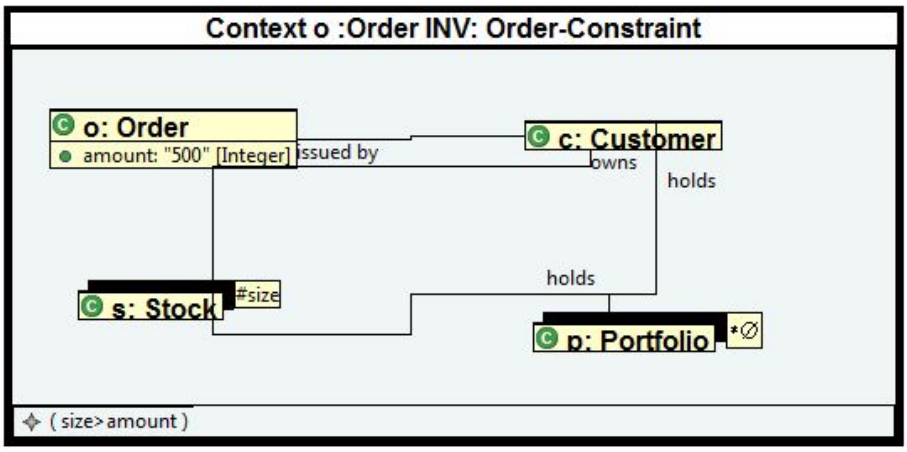


Figure 3.2: Constraints expressed in VOCL

The UML model uses different types of models to address structural and behavioural requirements. The models uses a vague packaging approach to separate concerns and then express their relation. The structure is expressed

[3] Version 11.4

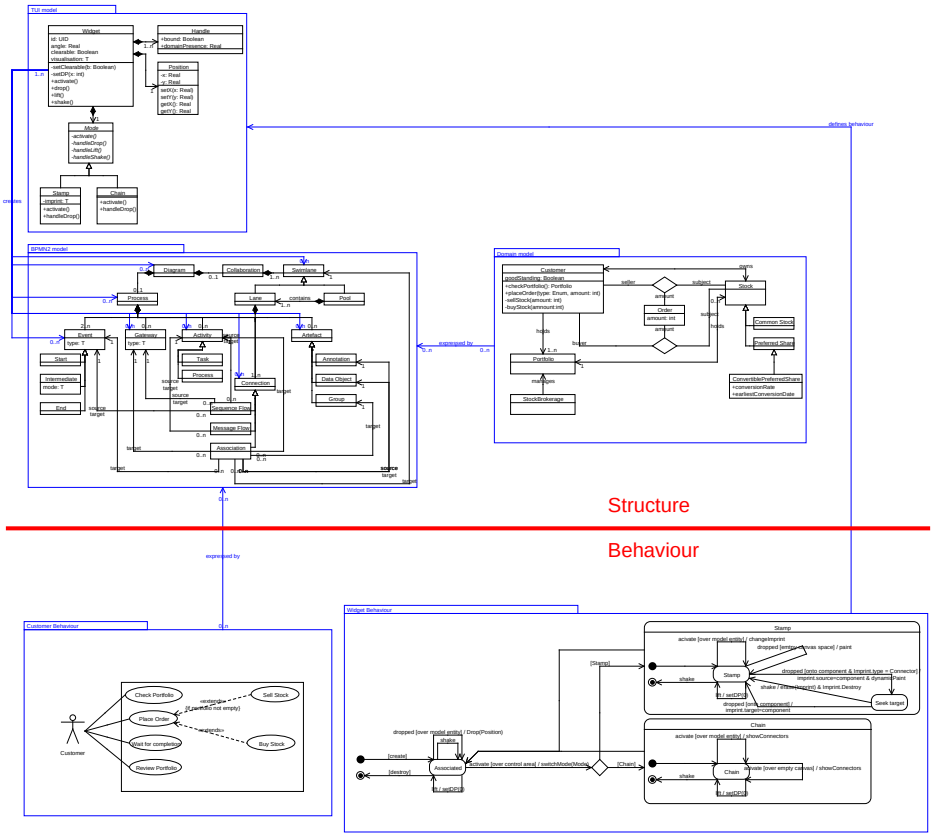


Figure 3.3: UML model for the scenario

using Class Diagrams. User behaviour is modelled using an Use Case Diagram. The system behaviour is visualised using a Statechart. VOCL uses a visual representation close to that used in Class Diagrams to express the underlying OCL constraint.

3.5.3.2 VCL

The scenario modelled in VCL made use of the Visual Contract Builder (VCB), a tool for modelling VCL using an Eclipse plugin from the University of Luxembourg. The tool is available for download at `http://vcl.gforge.uni.lu/`. VCB uses a packaging mechanism which allows for the separation of concerns while modelling. VCL Structural Diagrams in the final stages for the TUI, BPMN2 and scenario can be found in Figure 3.4, 3.5, or 3.6 respectively.

The behaviour in VLC is modelled using Behavioural diagrams. Figure

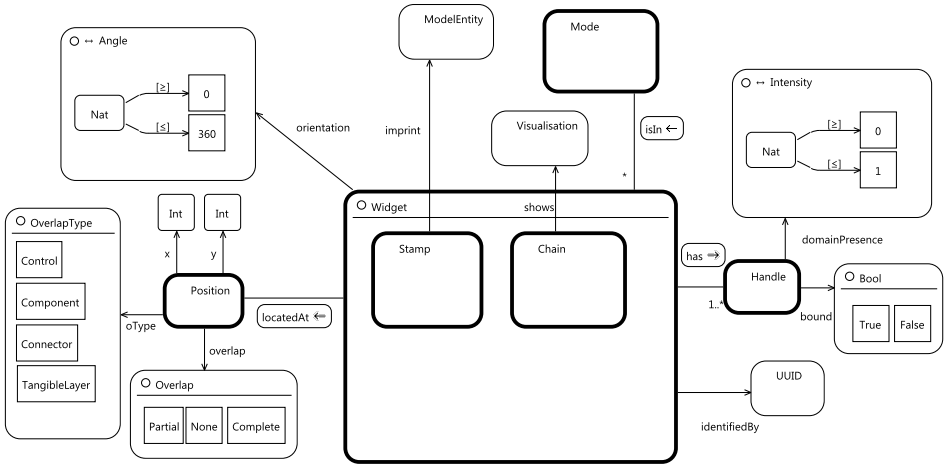


Figure 3.4: VCL Structure Diagram of the TUI model

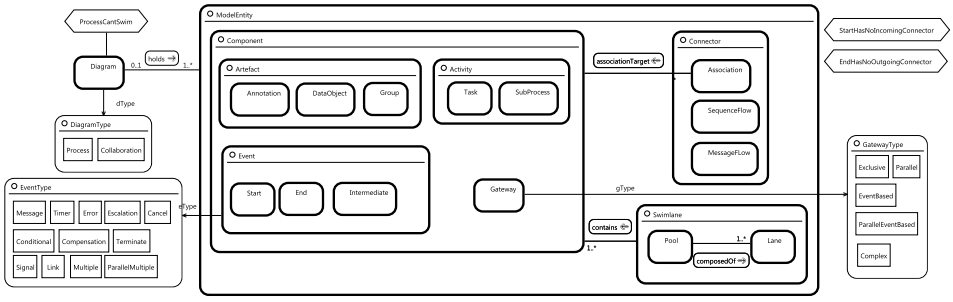


Figure 3.5: VCL Structure Diagram of the BPMN2 model

3.7 and 3.8 show these diagrams for the TUI and the scenario. The behaviour is expressed by refining the atomic behaviour such as the Stamp's Activate operation as shown in Figure 3.9. Update operations, those possibly producing state changes, are represented by contracts while query operations, those not modifying any states, are represented using assertions.

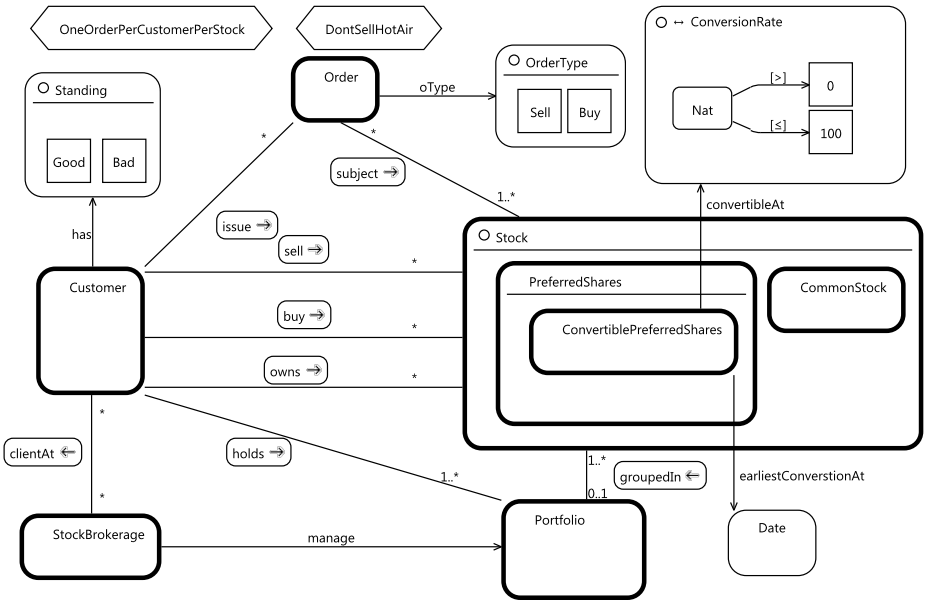


Figure 3.6: VCL Structure Diagram of the scenario

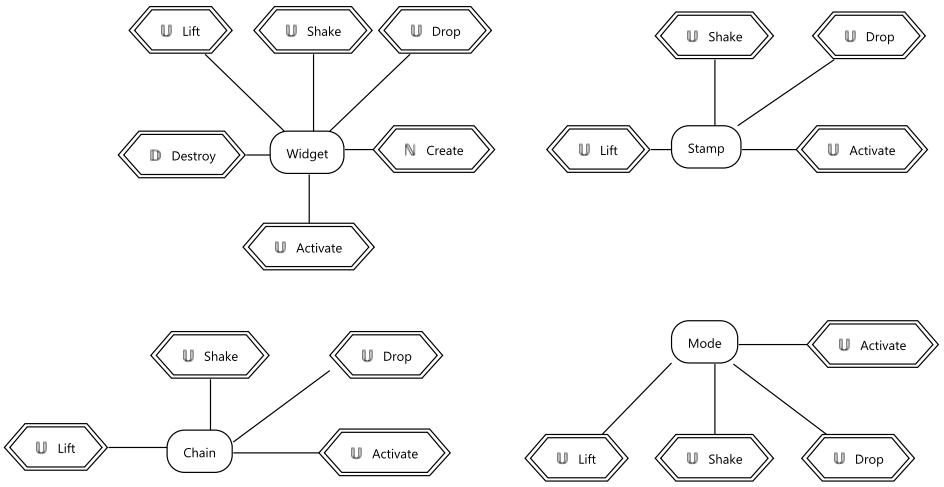


Figure 3.7: VCL Behaviour Diagram of the TUI model

Constraints, such as expressed by the invariants represented by the hexagonal boxes on the structural diagrams, are modelled using assertion diagrams. Figure 3.10 for example shows the invariant that expressed the implicit constraint of BPMN2 models that Process Diagrams cannot hold Swimlanes which can only be used in Collaboration Diagrams.

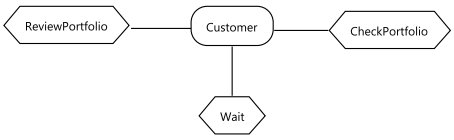


Figure 3.8: VCL Behaviour Diagram of the scenario

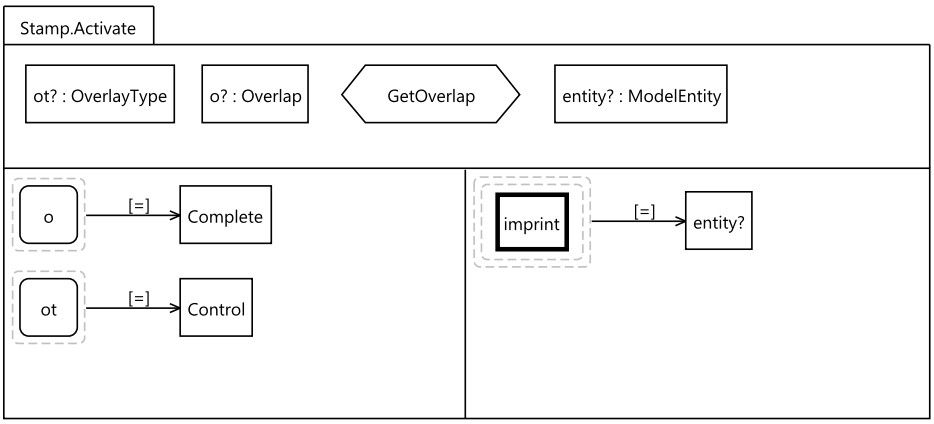


Figure 3.9: VCL Contract for the Stamp's local Activate operation.

3.5.3.3 Augmented Constraint Diagrams

Initially, a tool was used to attempt to model the scenario using Constraint Diagrams. The tool was the result of a joint project by the Universities of Kent and Brighton in the scope of a project to; *"[...]investigat[e] reasoning systems and tool for various visual constraint notations based on Venn and Euler diagrams"*. The tool as well as more information on the project, including the previous citation, can be found at `http://www.cs.kent.ac.uk/projects/rwd/`.

The tool worked fine for small problem instances but with the model growing beyond even a handful of contours, the tool proved unusable due to an exponential increase in computing time to make even the simplest modifications. The problem persisting in all compatibility modes, it rendered the tool useless. Together with the fact that the tool had no support for the augmented language dialect, a more archaic form of modelling was adopted; drawing by hand. The hand drawing were later replaced by models created in Microsoft Visio 2010 with custom stencils.

The problem with drafting models by hand or with simple drawing tool

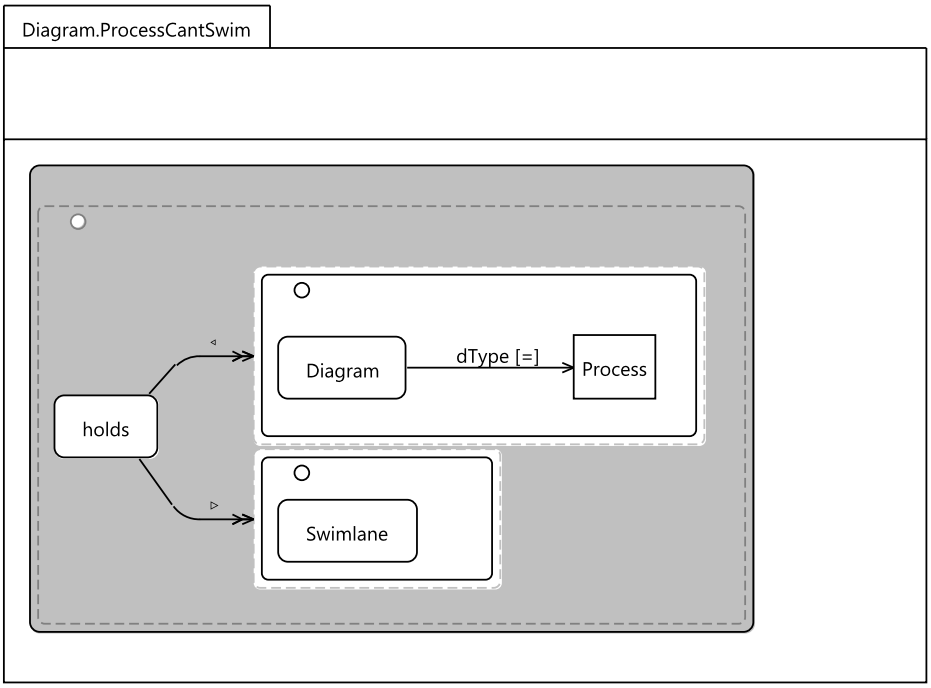


Figure 3.10: VCL Assertion of the ProcessCantSwim invariant

is that there is no mean to verify the model. This posed an obstacle that was rather hard to overcome as ACD experts are hard to find in a short time. Another issue was the tidiness of the model. In order to avoid multiple lines crossing and a loss in readability, some concepts were simplified. Figure 3.11 shows the BPMN2 and TUI model using the ACD notation. Figure 3.12 shows the modelling attempts to bring the TUI and BPMN2 models together to define how to use TUI instances to model BPMN2 instances. It also shows how behaviour is expressed using the augmented Constraint Diagrams language.

Both diagrams use the notation defined by [53]. The rectangle labelled *Diagram* in the centre of Figure 3.11 for example denotes the type *Diagram* which can hold either a Collaboration or a Process. This is expressed by the sets placed inside the type. The disjointness is implicit by the sets not overlapping. The fact that a *Diagram* can only hold either a Collaboration or a Process is noted by the spider that has one foot in each set. The type being shaded means that the type is not defined for anything not belonging to one of the named sets therein.

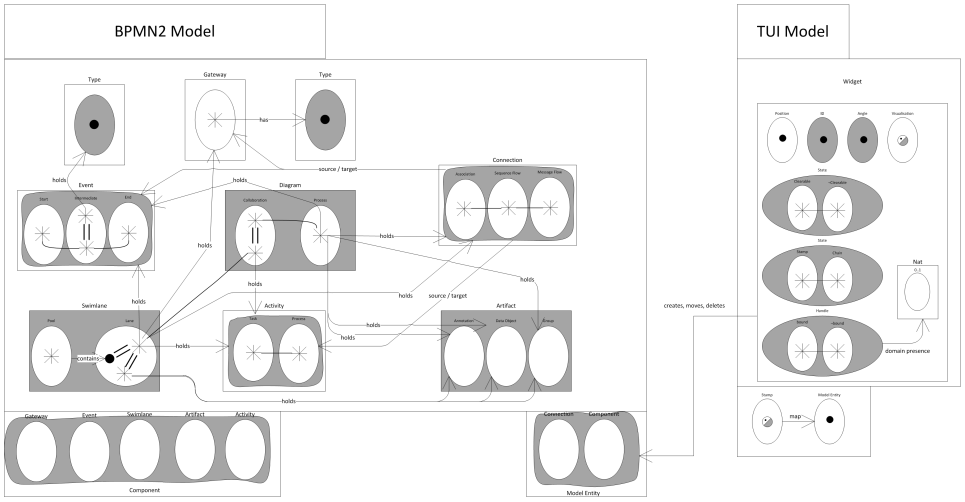


Figure 3.11: BPMN2 and TUI model using augmented Constraint Diagrams

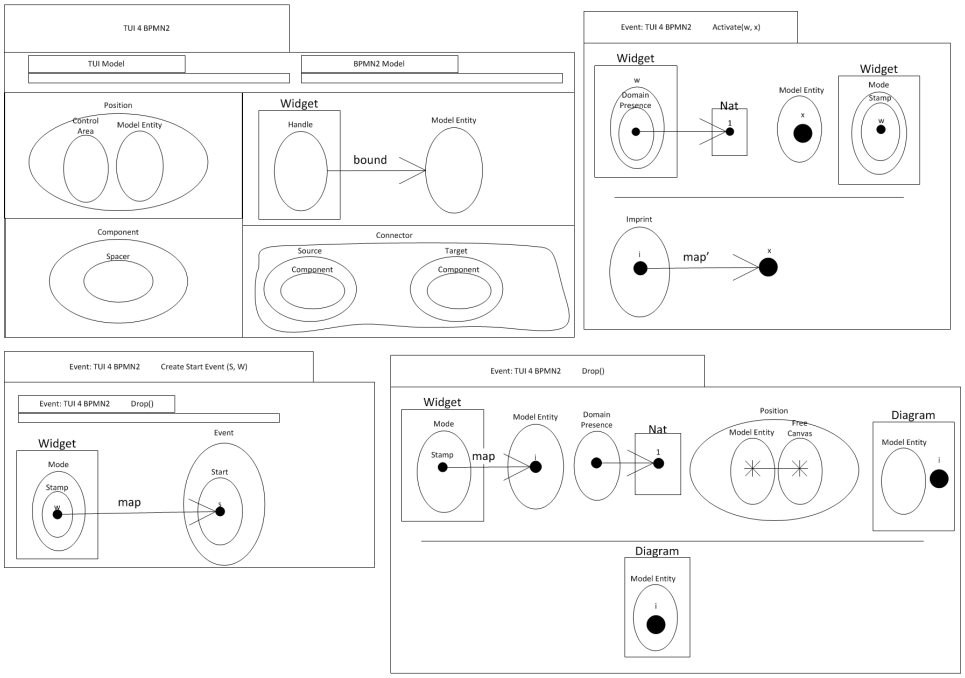


Figure 3.12: Behaviour and extension of concepts using augmented Constraint Diagrams

The behaviour, as shown in Figure 3.12 is given by the precondition, noted on top in the rectangle, and the postcondition, noted under the divid-

ing line. What is unclear is how *"if-clause"* preconditions are formulated. The model assumes that only the behaviour whose precondition holds is executed and should multiple definitions exist for the same behaviour, that all are verified and those with satisfied preconditions are executed.

3.6 Evaluation

With the study completed, only its evaluation remained to be able to reach a final verdict. In this section, the measurement criteria briefly described in Sub-section 3.5.1 are explained in depth. Each of these measures is rated on a scale, shown in Figure 3.13, and coloured for the reader's convenience. Furthermore, a weighting scheme will be defined. It has been drafted by sending a questionnaire asking business experts and people working in the Software Engineering domain on a five point Likert scale [105] how they would rate each criterion.

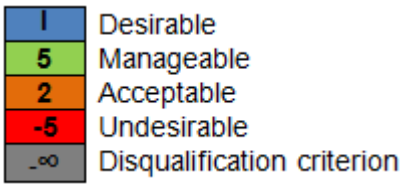


Figure 3.13: Coloured measurement scale

3.6.1 Tool support

Automation of model transformation is only one of many advantages that tool support has. Without tool support there is a strong possibility that other key points like error checking are not possible or at least more cumbersome. Also, the lack of an adequate tool usually means an increase in development time and having to make compromises when it comes to representing the model in a visual format or an increase in effort required to deliver the same result. In order to evaluate the suitability of a tool, the following points are observed and evaluated.

3.6.1.1 Availability

This criterion gauges whether tool support is available.

Measure	Meaning
Yes	Tool support is available by either a first or third party tool.
No	No tool support is available at all.

3.6.1.2 Maintenance

This criterion gauges whether an existing tool is supported by a team and maintained. A tool for which no information has been released for at least a year is considered to be no longer supported.

Measure	Meaning
Yes	The tool is supported on a regular basis.
Partial	The tool is supported at irregular intervals.
No	Support or maintenance has been cancelled.
/	No tool exists to base this measure on.

3.6.1.3 Latest version

This criterion simply collects data about the version of the tool for documentation purposes.

Measure	Meaning
Pre-Alpha	The tool is in a very early stage of development and not officially available. Pre-Alpha builds include evaluation copies handed to probable investors.
Alpha	The tool is officially announced and the current version is aimed to draw attention rather than providing full functionality.
Beta	The tool is on the final stretch before release. Some bugs remain but the functionality is almost completely available.
Gold	The tool has been officially released and the maintenance lifecycle has begun. All functionality is included with this release.
/	No tool exists to base this measure on.

3.6.1.4 Branch

This criterion states branch that spawned the tool. This might be an interesting factor to judge the continuity plans for the tool and future versions, improvements and overall support.

Measure	Meaning
Academic	The tool was the result or by-product of academic research.
Research	The tool was the result or by-product of a research or standardisation effort by a research or special interest group.
Industrial - Open Source	The tool was the result of an industrial effort to generate revenue. The source code is available.
Industrial - Commercial	The tool was the result of an industrial effort to generate revenue. The source code is not available.
/	No tool exists to base this measure on.

3.6.2 Semantics & Transformation

In order to be able to transform a model into another model, say for generating a data model, the former would need to be expressed in a formal way to allow for a meaningful and coherent mapping or transformation. A lack of formalism could lead to problems with reproducibility and reuse of the language and transformation or mapping schemes.

3.6.2.1 Formally defined

This data point tells us if the model is formally defined or not.

Measure
Yes
Partially
No

3.6.2.2 Transformability

This data point measures whether models produced by the language are easily transformable as such and if the transformation is complete.

Measure	Meaning
Yes	Models produced by this language are easily transformable.
Partial	Model transformations can only be done partially or require a lot of effort.
No	Models produced by this language are not transformable or require effort that would go beyond simply redoing the model in another language better suited for transforming models.

3.6.3 Expressivity

The expressivity criteria try to capture data points which measure the performance of the VMLs to be able to compare their ability to correctly model the scenario. They measure these criteria by category; structural requirements, behavioural requirements, or constraints. Therefore, each of the following sections is measured thrice with the exception of the ratio which is not observed but computed.

3.6.3.1 # X (# X)

This data point measures the number of requirements for category X. Note that the absence of colour indicates that this measure takes a simple integer value.

Measure	Meaning
Integer	The number of requirements in this category.

3.6.3.2 # X satisfied (# Sat_X)

This data point measures the number of requirements of each category that have been satisfied.

Measure
>= 90%
75% <= x < 90%
50% <= x < 75%
< 50%

3.6.3.3 # requirements partially satisfied (# Part_Sat)

This data point measures the number of partially fulfilled requirements.

Measure	Meaning
Integer	The number of partially fulfilled requirements in this category.

3.6.3.4 # unsatisfied requirements (# UnSat)

This data point measures the number of unsatisfied requirements.

Measure	Meaning
Integer	The number of unfulfilled requirements in this category.

3.6.3.5 Ratio

This data point computes the ratio of satisfied requirements to the total number of requirements by Ratio $= \frac{\sum_{X} \#Sat\_X}{\sum_{X} \#X}$

Measure
1
0,9 <= x < 1
0,75 <= x < 0,9
0,5 <= x < 0,75
< 0,5

3.6.4 Usability

It is quite difficult to measure but important to know how easy it is to learn and use a new language. This is of course a highly subjective measure. Still, an objective approach can be tried using absolutes where possible in order to compare the languages with regard to usability.

3.6.4.1 Naming conventions

This measure states whether naming conventions are used and if to what degree. The adherence to naming conventions is important as it enables novel users to get used to the language and not confused by terminology.

Measure	Meaning
Core	Naming conventions are used for core language features only.
Complete	Naming conventions are used for core language features as well as any extensions the language offers.
Mixed	Naming conventions are sometimes followed and sometimes not.
None	No naming conventions are followed.

3.6.4.2 Naming fit

This measure captures whether names are well chosen and not misleading. Using easily comprehensible and meaningful names for concepts reinforces the effect of metaphors and abstractions.

Measure
Yes
Partially
No

3.6.4.3 Documentation

This measurement captures whether the language is documented and if then to what degree.

Measure	Meaning
Outline	The documentation only consists of a rough outline.
Core	The documentation details all core language features.
Complete	The documentation spans all language features.
None	These are not the droids you are looking for.

3.6.4.4 Tutorial

This data point captures if there is a tutorial and how extensive it is. Only disjoint examples which cover different scenarios are considered.

Measure
None
[1..3] examples
]3..5] examples
>5 examples

3.6.4.5 Hands-on tutorial

This data point captures if there is a hands-on tutorial and how extensive it is. Only disjoint examples which cover different scenarios are considered.

Measure
None
[1..3] examples
]3..5] examples
>5 examples

3.6.4.6 Primitive mutability

This measure notes if the use of language primitives depends on the context it is used in or if it is static.

Measure	Meaning
None	No mutability. All primitives are used in one context only and have only one meaning.
Low	Less than (inclusive) 10% of all primitives are mutable.
Medium	Less than 50% but more than 10% of all primitives are mutable.
High	More than (exclusive) 50% of all primitives are mutable.

3.6.4.7 Live suggestions

This criterion aims to capture the interactivity that the tool offers which might help users pilot the tool.

Measure	Meaning
Yes	The tool offers functionality akin to auto-completion and suggests components to the user based on the current context.
Partial	As above but the feature does not work for all components and in all situations.
No	No such functionality is provided by the tool.
/	No tool exists to base this measure on.

3.6.5 Error Checking

This criterion aims to evaluate the different levels of error checking the language might offer in conjunction with any tool there might be for the language.

3.6.5.1 Time

This data point notes at which kind of error checking is performed. Note that the first two options are only observable if the language is backed by a tool.

Measure	Meaning
Real-time	
Compile-Time	
External	The tool does not offer error checking but there are other automated ways to check for errors.
/	No error checking possible.

3.6.5.2 Syntax highlighting

This measure captures if an existing tool uses syntax highlighting to provide feedback to the user about syntactical errors.

Measure	Meaning
Yes	Full syntax highlighting
Partial	Syntax highlighting is only supported for some of the language's features.
No	
/	There is no tool.

3.6.5.3 Degree

This data point states to what degree a tool offers error checking facilities.

Measure	Meaning
Yes	
Partial	Error checking is only supported for some of the language's features.
No	
/	There is no tool.

3.6.5.4 Error correction suggestion

Tool support offers the means to suggest correct values to the user in case he should have made an error. This data point captures if such a functionality is offered by the tool.

Measure	Meaning
Yes	
Partial	Error correction suggestion is only supported for some of the language's features.
No	
/	There is no tool.

3.6.5.5 Debugging possible

This measure captures if an existing tool offers, and to what degree, to debug models.

Measure	Meaning
Yes	
Unavailable	Models cannot be instantiated by the tool to be debugged.
Partial	Debugging is only supported for some of the language's features.
No	
/	There is no tool.

3.6.6 Verification

This criterion aims to evaluate the possibility to validate the model against the requirements after it has been completed. For most languages this step will have to be made manually by the clients. The reason is simple, if the requirements were formal enough to be checked against a model than they could be used to produce said model in the first place. Nevertheless, the language can offer some tools to statically or dynamically verify the soundness of the model.

The basic option would be the verification by a static tool akin to the verification scheme of an XML document. The next best step would be the real time error checking that should guarantee the soundness of the final model. More advanced techniques then allow for instantiation of the model which makes it easier to visually see quirks and kinks in the model that are not in line with the requirements. The current pinnacle of functionality that can be provided is an interactive model explorer that does not only generate random model instances but allows the user to explore and build instances freely.

3.6.6.1 Modularity

This measure aims to capture the modularity of the language. Is there a clear way to implement separation of concerns through use of for example packages or simply using different files which need to be imported? Modularity is a desirable quality as it gives a clear option to separate concerns.

Measure	Meaning
Yes	The language is modular.
Partial	The language offers modularity up to a certain degree.
No	The language is not modular.

3.6.6.2 Verification scheme

This measure states which scheme the language or related tools offer to verify the model.

Measure	Meaning
Instantiation	Models generated by the language are instantiable using a provided tool.
Interactive instance	A tool offers the interactive exploration of model instances.
Manual	The model needs to be manually analysed with methodologies not part of the language.

3.6.7 Weighting scheme

In an attempt to judge if the subjectively picked evaluation criteria were adequate for the task and to inject more objectivity into the study, a questionnaire was devised. A sample of the questionnaire can be found in Appendix I. The returned questionnaires were evaluated by attributing one vote per item per intensity on the Likert scale [105]. *"Not important at all"* was worth one point whereas *"Very important"* was worth five points. The sum of votes for each category were then divided by the total number of votes, giving a normalised average per category. The result can be seen in Table 3.14.

The distribution of returned questionnaires was heavily skewed in favour of Software Engineers with only one fifth of all questionnaires stemming from business experts. This was not by design but only returned questionnaires could be evaluated. Furthermore, the response was quite poor with only 6 questionnaires being sent back in total.

3.7 Results

All results were compiled by the coloured scale as defined by Figure 3.13. In order to facilitate the computation of the final results, an Excel table was used. Table 3.15 shows the findings of the study. For a list of requirements

Weighting		
Category	**Votes**	**Weight**
Tool support	19	0,1496
Expressivity	23	0,1811
S & T	22	0,1732
Usability	22	0,1732
Error Checking	20	0,1575
Verification	21	0,1654
Sum	127	1

Figure 3.14: Table of votes and normalised weights

that were considered for the expressivity criterion, consult Appendix G. As can be seen in the table, the language with the highest score is the Visual Constraint Language. The following paragraphs will comment on the results and highlight interesting findings.

Taking a close look at the *"Tool support"* criteria, one major difference between all tools can be identified, the degree of maintenance. While all languages have at least one tool, only the one offered for VCL is being maintained. All other tools were the results of projects that have since concluded and are, hence, no longer maintained. With VCB, the tool for VCL, also hailing from the academic sector, it might share the same fate if the language fails to reach the critical mass to persist beyond the current project. Please note that for the language duo UML&VOCL, only tools for VOCL were evaluated as UML offers a myriad tools in various forms and more than enough were adequate for the task. This is a major benefit if one uses an accepted standard.

The *"Expressivity"* criteria offer little room for interpretation. All requirements that had been dressed have completely been satisfied. However, with tools offering no support regarding error correction or highlighting for VOCL and ACD, the result could include mistakes. Therefore, those languages are given the benefit of the doubt, which means allocating full points. With the VCB tool offering live error checking, less mistakes have probably been made in VCL although a final verification by an expert would still be advisable.

Tool support		UML2 + VOCL	Augmented Constraint Diagrams	VCL
	Availability	10	10	10
	Maintenance	-5	-5	5
	Latest version	10	10	10
	Branch	5	5	5
Expressivity				
	# Structural requirements	27	27	27
	# Behavioural requirements	11	11	11
	# Constraints	4	4	4
	# Structural requirements satisfied	10	10	10
	# Behavioural requirements satisfied	10	10	10
	# Constraints satisfied	10	10	10
	# Requirements partially satisfied	0	0	0
	# Unsatisfied requirements	0	0	0
	Ratio	10	10	10
Semantics & Transformation				
	Formally defined	2	10	10
	Transformability	2	2	10
Usability				
	Naming conventions	10	10	10
	Naming fit	10	5	5
	Documentation	5	10	5
	Tutorial	2	2	2
	Hands-on tutorial	2	2	2
	Primitive mutabiltiy	5	5	5
	Live suggestions	2	2	10
Error Checking				
	Time	-5	-5	10
	Syntax highlighting	5	-5	10
	Error checking	2	-5	2
	Error correction suggestion	2	-5	2
	Debugging possible	-5	-5	-5
Verification				
	Modularity	10	5	10
	Verification Scheme	2	2	2
Weighted total		18,99212598	15,77165354	26,92913386

Figure 3.15: Table summarising the results

"Semantics & Transformation" was straight forward to evaluate. With UML under criticism for not being thoroughly formal [45] it also did not reach the full score for transformability. While some diagram types can without much effort be transformed, this cannot be said of all of them. While the core Constraint Diagrams can be translated into OCL and then further, the augmented version lacks that possibility.

While UML is very well documented and offers a huge amount of tutorials spanning all its aspects, VOCL is lacking in those disciplines. While only very few tutorials exist, the documentation is usable but not complete. The naming conventions and fit on the other hand leave nothing to be desired. The augmented version of the Constraint Diagrams are well documented and offer a couple of tutorials. However, the language uses some rather mathematical or domain specific terminology which might complicate matters for

non-experts in the field. The same holds true for VCL. While the language is well documented, the VCB could offer more stand-alone documentation even though the build-in suggestion system makes up for it to some degree.

"Error Checking" criteria are highly dependent on the use of an automated process or tool. With the tool for the augmented Constraint Diagrams failing to work properly and the initial models having been drafted by hand, there were no error checking facilities. Only very few UML tools offer error checking and so does the VOCL tool. The conversion to OCL allows for the use of some error finding methodology and the tool itself restricts the syntactical constructs that can be visually expressed thereby somehow forcing the user to not make mistakes. VCB offers real rime error checking and details about the nature of the error for most of the diagram types. None of the languages or their tools can instantiate models which would have helped in the validation and verification process.

VOCL benefits from UMLs packaging facilities to separate concerns and express models in a modular fashion. VCL also applies a packaging scheme. While it is possible to extend diagrams in the augmented Constraint Diagrams, thereby separating concerns, it seems more cumbersome than the packaging mechanisms offered by the other two languages. Without any possibility to instantiate the models, all models produced by the languages will have to be verified manually.

3.8 Study Conclusion

The author would like to not that while this study follows a quantitative-experimental approach, all choices were subjective and although the author tried to make choices as objective as possible, it is by the limited scope and time not possible to conduct a proper objective study on the matter at hand. Noteworthy flaws and kinks are the picking of measurements which is a highly subjective activity and the lack of experts to validate all models or draft the models in the first place. Readers are invited to determine the validity of the study for themselves. The following conclusion is only to be viewed in regard of the shortcomings of the study and should not be seen as a general statement or ranking of the VML under study.

After an initial broad selection of all related material, only a good two dozen resources were deemed relevant. The thirteen resources that remained after further selection could be grouped in three camps, one routing for the Visual Object Constraint Language (VOCL), one for Constraint Diagrams and one for the Visual Constraint Language (VCL). VOCL required the use of UML and Constraint Diagrams, not expressive enough on their own, were

used in an extended version, Augmented Constraint Diagrams (ACD). VCL was used as is. After selecting adequate tools for the modelling process, the different advantages and disadvantages of the languages and their tools became clear.

With all tools except for the Visual Constraint Builder (VCB), VCL's tool, being either unusable or not up to speed with that modern tools offer in regard of usability and support, VCL was the clear winner in that regard. UML in conjunction with VOCL proved adequate to model the BPMN2 Stock Management scenario. The use of VOCL however was rather cumbersome and error prone. Modelling using the ACD easily and quickly advanced which was due to the fact that it was done by hand. Unfortunately this also meant no support or error checking. With the language not being broadly established and having no expert at hand it has yet to be determined if the produced model is correct. With VCL, the modelling process proved to be very modular with the language separating structure, behaviour and constraints clearly even from a modelling point of view.

Having gathered all measurements and weighted the scores for each evaluation criteria, the ACD took third place mainly due to the lack of tool support and hence error checking capabilities. The effects on productivity were not measured although the drafting of models by hand on paper could prove to be problematic in a collaborative industrial or business environment. Second place was claimed by the tandem of UML and VOCL. While UML performed admirably, there is a reason why it is so successful after all, VOCL was not able to propel the team forward. Answering RQ2 1.1, the GPVML that performs best in modelling a VML scenario for use on TUI; is VCL. It offers a well defined language that could offer a better naming scheme but offers very little to be desired after the brief study. The VCB tool did help enormously in cementing VCL's gold medal but without continued support for the tool the edge that VCL currently holds over the other languages might quickly become dull.

Chapter 4

The Visual Contract Language, an introduction

The Visual Constraint Language (VCL) [50, 97, 98, 99, 106] is a novel visual modelling language. Section 3.3.2 introduced some of the basics which will be further elaborated on in the following paragraphs. VCL is used for the formal specification of requirements in software systems. The structural and behavioural requirements as well as contracts on the behaviour and invariants are all expressed in VCL using its own formal syntax. Its visual syntax is inspired by Higraphs [4], UML class diagrams [49] and Entity-Relationship diagrams [54]. VCL's formal semantics are a result of the work of VCL's inceptor, Nuno Amálio, on ZOO [107, 108], a semantic domain for the object-oriented paradigm expressed in the Z specification langauge [109, 110]. A Z specification can be generated for a VCL model for the purpose of verification and validation. A Z specification generated for a VCL model is given in [106].

VCL embraces many of the concepts thought to be desirable for varying aspects of Software Engineering. VCL is a modular language, implementing the principle of *Separation of Concerns* [111], thought to increase readability by reducing complexity which in turn also eases maintainability. VCL's approach on modularity also allows for addressing vertically distributed concerns frequently found in complex systems [106]. Modularity in VLC is achieved by using a packaging scheme at the structural and behavioural level and by using contracts and constraints on a local level. By using contracts and constraints to express local concerns and specify the behaviour of components, VCL subscribes to the *Design by Contract* [103] principle.

This chapter elaborates on VCL. The reader is referred to [97, 106] and the tutorials available on the VCL web-site [1] for further information.

[1] `http://vcl.gforge.uni.lu/tutorials.html`

4.1 Syntax

VCL uses several primitives and diagram types. The following sections introduces VCL primitives in Section 4.1.1, Structural diagrams in Section 4.1.2, Behavioural diagrams in Section 4.1.3, Package diagrams in Section 4.1.4 and lastly, Assertion and Contract diagrams in Section 4.1.5. The following paragraphs were compiled from [106], adding newly added syntactical elements that are missing in the report.

4.1.1 Primitives

Diagrams are built using primitives. While all the primitives have a core meaning, they vary slightly depending on the diagram they are used in. The following paragraphs will go through all primitives used in this document. For a complete view of all primitives, consult [106].

Packages A VCL model is organised around packages, which are represented visually as clouds. Packages can be of two kinds, containers or ensembles. A container package merely groups concepts and their local behaviour. Like containers, ensemble packages also group concepts and their local behaviour, but in addition they can define their own global behaviour, which typically, involves a coordination of its concepts. In package diagrams, insideness is used to express that all concepts defined in the contained packages are grouped, and exposed through one single package. This is useful to structure concepts. Figure 4.1 shows a container package. An ensemble package containing a container package is shown in Figure 4.2.

Figure 4.1: A simple VCL container package

Figure 4.2: An ensemble package containing another package

Blobs Blobs, a name coined by Harel in [4], are rounded shapes used to denote sets and subsets. The notation stems from Euler and is similar to

that used in Set Theory. However, alike to Harel's Higraphs, overlaps are not permitted as the topological information carries formal meaning. Figure 4.3 shows the blob `String`, `UUID` and `Accessor` with the latter two being disjoint subsets of `String`. Note that the insideness relation must be acyclic which is enforced locally by the topology but must be respected globally as well.

Figure 4.4 shows a blob defined by its contents, denoted by the ◯ symbol. A `Bool` is exclusively defined by the objects, see 4.1.1, `True` and `False`. Figure 4.5 shows a reference blob used to refer to a remotely defined blob by the symbol ↑. In this case to `String` defined in the package *P*. See the previous paragraph for information on packages.

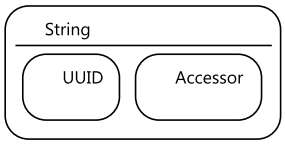


Figure 4.3: VLC blobs illustrating insideness

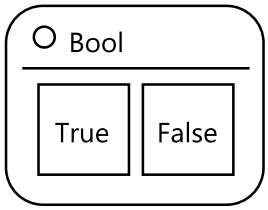


Figure 4.4: A blob defined by its contents

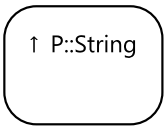


Figure 4.5: A blob referencing a remote blob

Objects Objects are rectangular shapes that are used to denote elements of a set. They are commonly used for constants. Apart from their name, they carry the label of the set they belong to. Figure 4.6 shows the constant `xPath`, an element from the set of `URI` as defined in the *P* package.

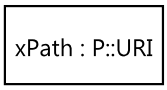


Figure 4.6: An object

Edges Edges connect packages, blobs, and objects. Property edges as shown in Figure 4.7 visualise properties possessed by all members of the set connected to the edge's source. The head denotes what "type" of property they have and the label indicates the name. Figure 4.8 shows a relational edge. They define a new set of tuples. The relational set might be constrained by cardinalities other than the regular one-on-one mapping. Only blobs denoting sets can be connected by relational edges. Origin edges, shown in Figure 4.9 reduce the scope of attached assertions or contracts to

be local to the blob of origin.

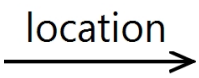


Figure 4.7: A property edge

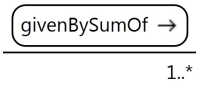


Figure 4.8: A relational edge

Figure 4.9: An origin edge

Package edges, as in Figure 4.10 expand visibility to primitives defined in external packages. Merge edges, shown in Figure 4.11 are used to merge blobs from different package blobs, wearing the name of the to be merged blobs as label. Figure 4.12 depicts an override edge, overriding the definition of the blob with the name indicated on the label in the target packet with the definition from the origin packet.

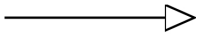


Figure 4.10: Package use

Figure 4.11: Merging blobs

Figure 4.12: Overriding blobs

Assertions As shown in Figure 4.13, assertions, formerly called constraints, are depicted by a hexagonal labelled shape. They observe the state of the system without modifying it. They can be used to describe constraints that must always be true, and query operations that have no restriction on their return value.

Figure 4.13: An assertion

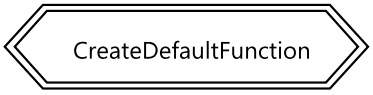


Figure 4.14: A contract

Contracts Contracts are double-lined, labelled hexagonal shapes as shown in Figure 4.14. They modify the system state according to the pre- and postconditions they define.

4.1.2 Structural diagrams

Structural diagrams (SDs), an example is shown in Figure 4.15, include blobs, objects, assertions as well as the edges linking those elements. Each

package of a VCL model has one SD, defining the package's state space. In SD, blobs may incorporate other blobs or objects, the first expressing the insideness property and the latter, together with the ◯ symbol on the blob, the definition of a concept by the sum of its values marked by the objects. Blobs can either be value blobs, a set of immutable values, or they can be domain blobs, characterised by the bolded border, that affect the state of the system as they are created and change state.

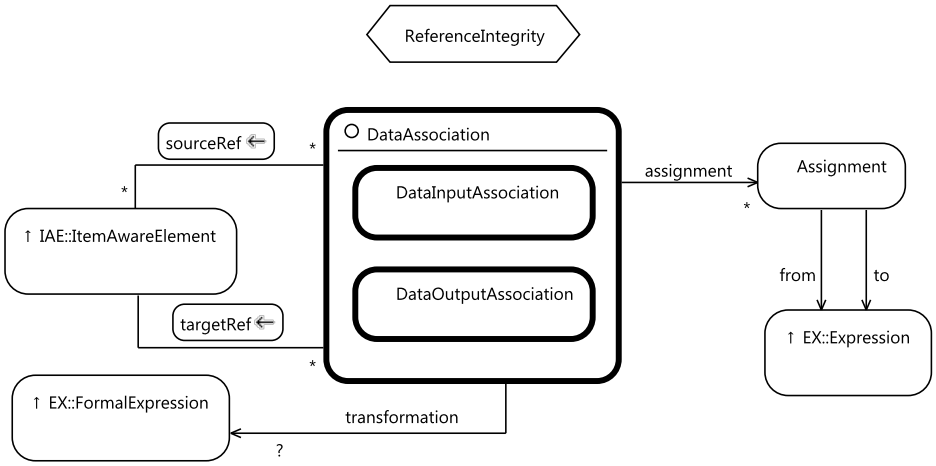


Figure 4.15: A VCL structure diagram

Figure 4.15 shows a structural diagram of a BPMN2 concept, namely `DataAssociations`. The concept will be elaborated on in Appendix B.18. The `DataAssociations` value blob defines two subsets through the topological layout. The property edge `assignment` models that a `DataAssociations` can have multiple `Assignments` which are predefined values given by a mapping expressed using `Expressions`, imported from the ***EX*** package. The `targetRef` relational edge creates a many-to-one mapping. The new set contains tuples of the type (`DataAssociations`, `ItemAwareElements`). The SD displays a constraint on `ReferenceIntegrity`.

4.1.3 Behavioural diagram

Behavioural diagrams (BDs) define a finite set of operations. These can be either local, if connected by an origin edge to a blob, or global if unconnected. Only global operations can be referred to in a global context. Furthermore, operations are either updating a set expressed through the double-lined border or observing a state. Local update operations come in three shapes;

- they can create new set elements, denote by a leading `N`,
- they can remove elements from a set, denoted by a leading `D`,
- they can update elements of a set, denoted by a leading `U`.

Global operations can do neither of the above as they cannot act upon the local context of a set. They have to call upon local operations to do so. Figure 4.16 shows the BD of the TUI metamodel concept of layers. Here, the global operation `AddWidget` calls upon the `Add` update operation of the `TangibleLayer` to add a `Widget`.

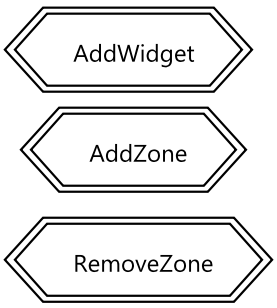


Figure 4.16: A VCL behaviour diagram

4.1.4 Package diagram

Package diagrams (PDs) feature only packages and edges. A PD can either incorporate other packages inside its cloud-like shape or import them. The latter can be seen in Figure 4.18. The imported packages are the target of a package use edge. Packages can receive an alias which is noted in the package blob. Importing packages exposes all elements contained in those packages for use in the global context of the current state. Enclosing packets within the contour of another package blob, as shown in Figure 4.17, expresses that all concepts from the enclosed package are incorporated into the state space of the enclosing package. If concepts are defined in multiple packets, each only partially modelling the concept, merging the concepts is necessary. A merger edge connects the packages where blobs need to be merged, labelled by the name of the blobs to merge. All properties and relations of the separate blobs are merged and exposed as one concept through the enclosing packet.

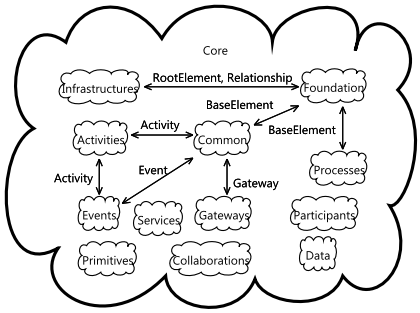


Figure 4.17: A package diagram illustrating merges and incorporations

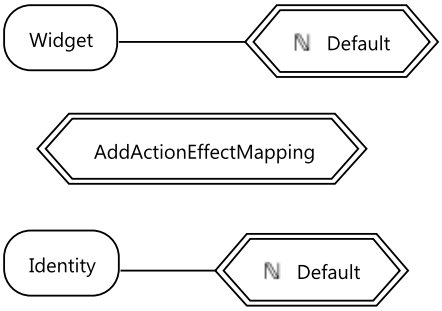


Figure 4.18: A package diagram illustrating imports

4.1.5 Assertion diagrams

Assertion diagrams (ADs) observe the system state, expressing constraints, invariants, or operations. The upper compartment of the AD, as shown in Figure 4.19, is used to declare variables locally or through imports. Variables suffixed with ? or ! denote inputs and outputs respectively. The import and passing of variables sometimes requires them to be renamed. This is done by rename lists on constraint labels in the declaration compartment. For example, renaming `w!` to `w?` would be written as `[w!/w?]`.

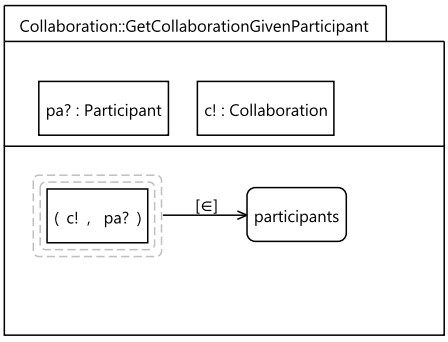


Figure 4.19: A VCL assertion diagram specifying an operation

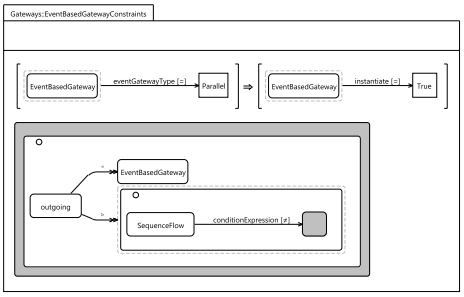


Figure 4.20: A VCL assertion diagram specifying constraints

The lower compartment is used to define the predicate of the assertion built from blobs, objects, relational and modifier edges as shown in Figure 4.20. Modifier edges are a variant of property edges. They can take on operators to apply operators found in Set Theory such as domain restrictions, $\lhd$, and domain range, $\rhd$. Shaded blobs denote the empty set, which can be used to say that, as seen in Figure 4.20, the set expressed through the contained predicate is empty. Hence, a simple shaded blob denotes the

empty set. Predicates use a notation borrowed from First Order Logic. Special operators such as the cardinality operator, #, can be used to form further predicates.

4.1.6 Contract diagrams

Contract diagrams (CDs) specify operations that alter the system state. They describe what preconditions must hold before the operation is executed in the lower left-hand compartment and what postconditions hold immediately after the operation has executed in the lower right-hand compartment. As was the case in ADs, CDs also contain a declaration compartment. The use of the latter one is analogous to that in ADs. Figure 4.21 shows an example of a CD. In addition to predicates, which were explained in the previous section, CDs make use of action units which indicate the object or set to change as a result of the operation.

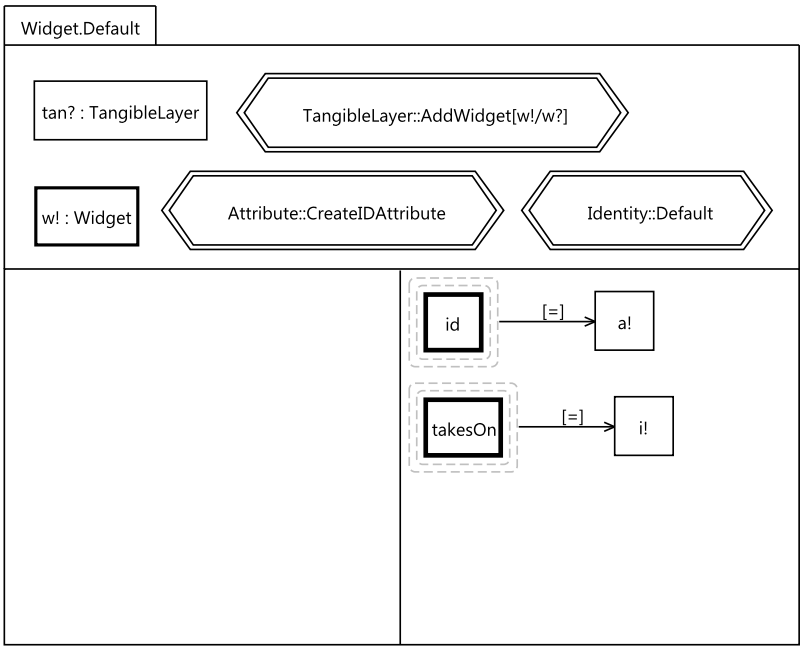


Figure 4.21: A VCL contract diagram specifying an operation

4.2 Semantics

VCL's semantics is defined using a translational approach [112] by translating VCL diagrams into the Z specification language. This gives VCL a solid grounding in Set Theory and predicate logic. As mentioned in [97, 106], further translations are planned.

4.3 Modelling in VCL, using VCB

Modelling in VCL has a few restrictions not enforced by the nature of the visual syntax and the editors. Most of them are minor inconveniences introduced by bugs. Bugs are fixed regularly and the team welcomes bug reports. This is to be expected of a language still being actively researched and developed. For example, during the modelling process, *value* could not be used as a label. Using a prefix such as an underscore solves the problem while still leaving the labels human-readable and easily identifiable.

Compared to VML like UML, VCL has very few built-in types. As of the time of writing, only naturals and integers are built-in by the NatBlob respectively IntBlob. Future versions will likely introduce more built-in types such as Booleans. Due to the limited number of types readily available, modelling in VCL usually requires the creation of a package to declare additional primitives for use in the system. Types are defined by creating value blobs as, usually, values are predefined and limited. For types with a large number of possibilities, such as strings, no specific values are defined. For types with a small amount of values or types for which their content is not clear by their name, values are added by adding objects into the contour of the blob. Figure 4.22 illustrates both examples.

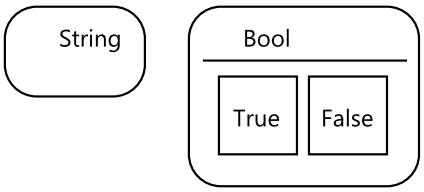


Figure 4.22: Declaration of additional types in VCL

Modelling VCL is easiest done using the Visual Contract Builder (VCB), a free Eclipse plug-in [2]. As of the time of writing, the tool requires Eclipse in at least version 3.6 [3]. The VCB download page also offer installation advice and two tutorials that introduce the basics of VCL. All reports and papers regarding VCL and cited here are also available on the site and offer a possibility to get acquainted with VCL beyond what is needed for reading this document.

VCB offers the possibility to create VCL packages, holding all diagrams defined by the package. The facilities are intuitive to use and offer a familiar look and feel as VCB has been built using the GMF framework [4]. VCB enables modelling of VCL through editors offering a palette with visual components as well as a context sensitive pop-up menu for available components. The context sensitive menu offers only syntactically valid model components. Modelling, especially in the learning phase, greatly benefits from the context-sensitive assistance of VCB. Another helpful aspect is the automated verification and validation that VCB offers. As the semantics of VCL are formally defined in Z, Z theorem provers are able to formally verify and validate VCL diagrams.

The VCB editors are built using a metamodel of the VCL notation. A partial metamodel is currently available in Alloy and UML [5]. In addition to the Z theorem prover, a VCB type checker will verify the well-formedness of VCL diagrams. Only well-formed diagrams will be translated into ZOO and hence checked by the theorem prover.

[2] `http://vcl.gforge.uni.lu/download.html`
[3] `http://www.eclipse.org/`
[4] `http://www.eclipse.org/modeling/gmp/`
[5] `http://vcl.gforge.uni.lu/metamodels/`

Chapter 5

A VCL Model of a TUI for modelling with BPMN2

5.1 Introduction

This chapter gives all technical and contextual details regarding Tangible User Interfaces (TUIs) and the Business Process Modeling Notation version 2 (BPMN2).Those will be detailed in Sections 5.2 and 5.3 respectively. Furthermore, the chapter will briefly introduce the metamodels of TUI and BPMN2 while deferring to Appendices A resp. B, for the VCL implementation of those metamodels.

Section 5.6 introduces the Business-to-Table concept, and its metamodel, which is thought to be part of the application that will allow to mode the ideation process using BPMN2 on a tabletop device. Thereafter, the models of the ideation scenario, Section 5.7, widgets, Section 5.8, and interactions, Section 5.9 will be discussed, as well as their modelling processes. All of the models mentioned above are drafts modelled using VCL and VCB. They can be found in, except for the interaction model, Appendices C, D, and E respectively.

This chapter answers the third research question by proposing the following intermediary questions:

RQ3 *Is it realistic to use a GPVML to model complex TUI applications?*

- **RQ3a** *Can concerns be separated in a meaningful way?*
- **RQ3b** *Can all requirements be modelled?*
- **RQ3c** *Is the expressiveness of VCL sufficient?*

Separation of Concerns (SoC) in Software Engineering is not only a good practice but also thought to help reduce complexity, increase comprehensiveness, facilitate reuse and ease maintainability[113, 111]. Separating concerns into modules or packages seems to be a practice in modelling too and it is not farfetched to think it will deliver the same or at least comparable advantages as mentioned in [114]. The answer to RQ3a will show if the SoC principle can be followed using VCL.

RQ3b is aimed as gauging the success of the modelling attempt. Should some requirements not be able to be modelled using VCL it could hint at some shortcomings or a lack in semantics. Identifying and documenting these will be essential to reach a conclusion to RQ3. RQ3c is somewhat related to RQ3b. If some pieces of the puzzle cannot be modelled it is likely due to the unavailability of some descriptive constructs. However, the answer to RQ3c will also allow capturing requirements unrelated to semantic shortcomings as well as some heavy semantic constructs that are hard to use. This research question is aimed at gathering some suggestions for the team working on VCL and to illustrate some future work and perspectives.

5.2 Tangible User Interfaces

This section details the TUI that is currently available at the Public Research Centre Henri Tudor in order to be able to understand the limitations of and requirements on the metamodel. Technical details that are omitted for the most part as they are not relevant in this context. All technical details are however included In Paul Bicheler's Master's Thesis [104]. His thesis also contains all details about the TUI Widget Toolkit (TWT) which serves as a source of requirements for the TUI metamodel.

5.2.1 TUI technical aspects

The table consists of a plexiglass-topped wooden box as seen in Figure 5.1 with a beamer projecting an image from below via a slanted mirror. Infrared light dispersers light the area below the acrylic glass surface so that the light reflected from the surface can be captured by an infrared camera. This process allows the reduction of noise considering the fiducial markers are black and white as can be seen in Figure 5.3. The user can move tangible widgets on the surface, unaware of what is going on below. This is important as it makes some of the TUI workings ubiquitous. In the scenario, tangible widgets or simply widgets, are composed of Sifteo cubes [115], a prototype of the Siftables MIT project [116], and a fiducial marker. The cubes without the considerably larger marker can be seen in Figure 5.2.

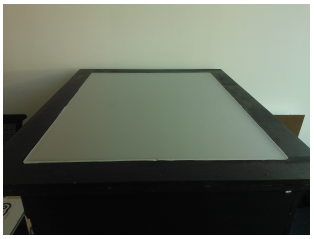


Figure 5.1: The TUI table.

Figure 5.2: Two Sifteo cubes on the table.

Figure 5.3: A fiducial marker.

5.2.2 A brief look at TWT

Paul Bicheler's thesis [104] on the toolkit for table-based tangible widgets serves as a requirement document to discuss the TUI metamodel in Section 5.4 and dressed in Appendix A. However, due to concurrent time frames of both theses, the latest version that has been considered for the metamodel did not include selector endpoints, function safety or system interactions. Also, at the moment the models were drafted, the application context held all layers. The following paragraphs will highlight some of the important aspects of the specification. However, the reader is invited to refer to Bicheler's Master's Thesis [104] for an exhaustive explanation on TWT.

Widget structure A widget's structure spans all digital and physical components of the widget. The topmost physical component is the handle. It is composed of other physical components giving the actual physical parts shape, feel and functionality by the means of actuators and sensors. Visual components are the digital representation of widget states. They can be directly bound to handles where they are visualised by the application to stipulate information or state changes by user interactions through the handle on the endpoints. They are a grouping term for widget components or zones, the latter of which is described in the paragraph about the application context below. Physical and digital properties are described by attributes.

Widget behaviour A widget's behaviour is given by a number of functions which contain a mapping of attributes from actions to optional effects and finally resolve in at least one endpoint. The creation of the function and the mapping of the attributes are made by toolkit users as is the goal of the toolkit. A widget has a default behaviour which is expressed through a default function with default function elements, default action and default

endpoint, to guarantee that every widget has a behaviour in addition to any behaviour the toolkit user might specify.

Actions are user input that is perceived by the system. Actions do not have input attributes as they always start a function. Their output attributes may be mapped to effects which can, depending on the attributes and current state, change the state and either proceed to another effect or an endpoint, depending on how the designer has mapped the attributes.

Mappings The mapping of the attributes of all function elements are dictated by the mappings associated with the function. A mapping specifies for each source attribute of a source function element a target attribute of a target function element. Actions or events can be continuous meaning that the system can update attributes and the effects respectively endpoints. The latter observe their input attributes and react once the new value of the attribute meats a precondition and is therefore relevant to the receiving function element. The toolkit specifies no constraints on the mapping of attributes other than type constraints which are not considered here. However, only attributes from action to effect, effect to effect, action to endpoint and effect to endpoint should be mapped. Furthermore, the source attribute should belong to the function element from the `mapFrom` relation and, symmetrically, target attributes should be properties of function elements from the `mapTo` relation.

Identity The identity of a widget gives its functionality. The identity, in a way, is the bundled functions that the widgets have, the sum of its behaviour. This abstraction makes it easier to design and talk about widgets. For example, a widget with the *Stamp* identity has a number of functions that give it the functionality to take on a given imprint and then create it on the canvas by dropping the handle onto the surface. This requires multiple functions related to dropping and lifting the handle, activating the widget and shaking it. Yet users who are working with the system can simply refer to it by its identity and it becomes clear what the associated behaviour is.

Endpoint The concept of endpoints is used to group zones and widget components as they can both be the target of user interaction. Zones are exposed through the application context and will be addressed in a later paragraph. Widget components can be the target of interactions in that they can be tasked to display or forward feedback to the user. To map user interactions to endpoints, they figure in functions as a required final element, wrapping up the function and, depending on the endpoints concrete type, finalise the user manipulation.

Binding The binding is a symmetric relation from one endpoint to another. Bindings are used for example to attach a visual component to a zone. A binding effectively merges both endpoints such that they become a new endpoint.

Selection A handle can select an endpoint. A selection is a less strict binding but still symmetric. Selections are used to temporarily select one or multiple components in order to act on those components or forward an action to them. For example, in order to delete an unbound entity present on the table, it must first be selected.

Application context The application context as described by Bicheler is split into zones and layers. It is thought that the application context is the interface between the widgets and the functionality they offer and the application they are used in. The application should make a distinction between two layers;

- the tangible layer is the uppermost layer. It effectively contains the widgets and exposes all of them to the application,
- the canvas layer containing all zones provided by the application. It contains the data representing virtual objects which are cast into zones and exposed for the widgets on the tangible layer to interact with.

Zones are areas of interest that are exposed by the application context for interaction with widgets. They are polygons wrapping data and commonly their visual component. For example, a zone might correspond to a box or an arrow. Interactions with zones will take effect on the entirety of the zone. Therefore it is essential for the application to manage zones correctly in regard to user intent and interest. Widgets interface with the application solely through the exposed zones.

5.3 BPMN2 and its usage in this document

The *Business Process Model and Notation, Version 2* (BPMN2) as defined by the Object Management Group [23] in 2011 is the second iteration of the, newly renamed, Business Process Modeling Notation which was originally published back in 2008. The language specified through the metamodel given by the specification document enables the drafting of high level business process models. The notation is mainly graphical and similar to that used in flowcharts. The metamodel describes all abstract concepts needed to build BPMN2 models.

The aim of BPMN2 is to provide an easy to use language that is intuitive for business practitioners to use yet provides a semantically sound

environment to work with, bridging the gap between the semi-formal implementation of business processes and architectures and the definition of high level and abstract business concepts. BPMN2 makes use of widely known components and naming conventions, therefore enticing a sense of familiarity.

BPMN2 elements exposed to the modeller comprise, for example, flow object, flows, swim lanes and artifacts. Flow objects are the main components of the model, activities, events and gateways. Those are interconnected using mainly sequence or messages flows. To organise different processes and structure different elements, pools and lanes are used. Most of these elements can be enriched with artifacts like annotations at the modeller's leisure. This small set of model elements, paired with their mutability in semantics, is what makes the language easy to grasp and use.

The BPMN2 specification document [23] was published by the Object Management Group in January 2011. The document spans, apart from the elements mentioned above, all other relevant aspects of the notation. It defines not only its metamodel but also conformance, symbolism, exchange formats and mapping to other languages such as WS-BPEL [1].

Element	Chapter	Reason
Conversations	Multiple	Lack of application from practitioners
Choreographies	Chapter 11	Lack of application from practitioners
BPMN Notation and Diagrams	Chapter 12	Low to no impact on the modelling activity
BPMN Execution Semantics	Chapter 13	Low to no impact on the modelling activity
XML Schemata	Multiple	Low to no impact on the modelling activity
Visual requirements	Multiple	No impact on the modelling activity
Visual constraints	Multiple	No impact on the modelling activity

Table 5.1: Limitations and reduction of the scope given by [23]

However, the specification document is very large with over five hundred pages. To model all of it would surpass the time constraints of a master's thesis by far. Therefore, only parts of the document could be considered and

[1] Web Services Business Process Execution Language also referred to as WSBPEL

from those, again, only parts modelled. Firstly, only the chapters relevant to the modelling activity have been considered, namely chapters seven to thirteen. This first slice of the specification was subsequently broken down further by relevance and use. Not modelled were the elements in the Table 5.1 for the reasons given. However, most of those elements were still studied before being discarded to make an informed choice. Section 5.5 will discuss the important aspects of the BPMN2 VCL metamodel dressed in Appendix B.

5.4 The TUI VCL metamodel

This section briefly describes the packages that make the TUI metamodel, discussing the modelling choices regarding the specification, the TWT framework introduced in [104] and briefly presented in Section 5.2.2. The metamodel has been designed using VCL. All VCL diagrams can be found in Appendix A. The metamodel has been split into separate packages, each addressing one specific concern. The appendix also includes a list of all VCL packages and their diagrams in Table A.1. Figure 5.4 shows the *TUI's* package diagram, giving the reader an idea about the structure of the metamodel.

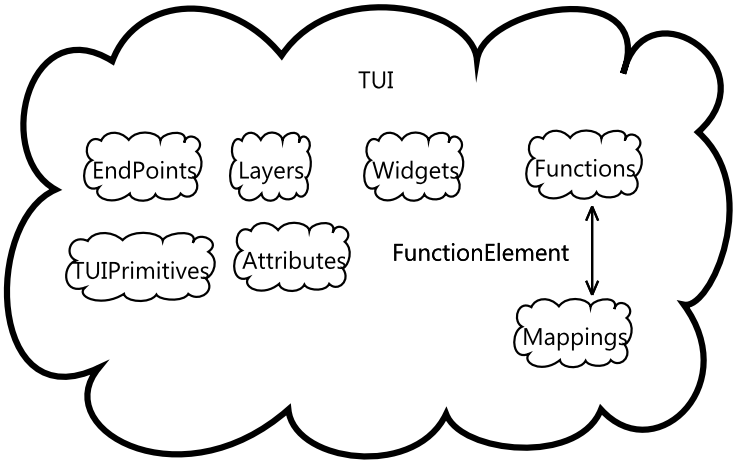


Figure 5.4: The *TUI's* package diagram

5.4.1 Packages

The following paragraphs detail all VCL packages for the TUI metamodel using a bottom-up approach.

TUIPrimitives The *TUIPrimitives* package defines all basic types found in the specification document that are not built-in in VCL. This package is used by most packages defining the TUI metamodel. To illustrate structural diagrams in VCL, the SD of this package as well as a detailed explanation is given below. These can also be found in Appendix A.1 for further details.

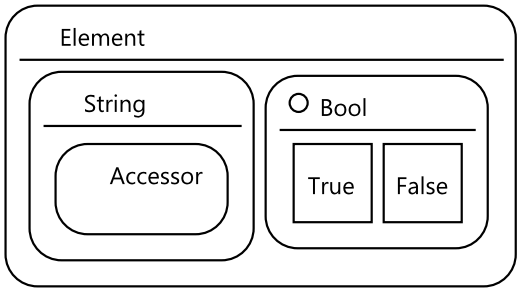


Figure 5.5: The *TUIPrimitives'* structure diagram

The SD found in Figure 5.5 specifies a boolean primitive, `Bool` which can take the values `True` and `False`. It also specifies `Strings` and `Accessors`, a subset of `Strings` as shown by the insideness on the SD. `Strings` in turn are a subset of `Element`.

Actions The *Actions* package deals with widget triggers, user input that is forwarded by the system and processed by one or more `Functions`. `Actions` model part of the widget's behaviour. To illustrate the use of the remaining VCL elements, sample explanations and diagrams are given below. For the complete explanation, see Appendix A.2.

The `Action's` PD shows the import edges towards four different packages and their alias definitions. For example, the *Effects* package is imported and aliased as *EFF* for use within the *Action* package.

An `Action` evaluates a `Condition`. An `Action` may have `Attributes`, one of which must be an `id`. `Actions` can trigger `Effects` or immediately produce a response in an `EndPoint`. `Actions` also include a reference to one or several `Mappings`, mapping their `Attributes` to either `Effects` or `EndPoints`. The `Action` specifies a local invariant, `EndPointRequired` which is

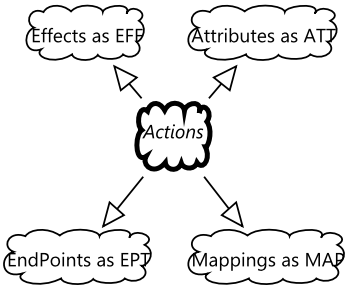


Figure 5.6: The `Actions'` package diagram

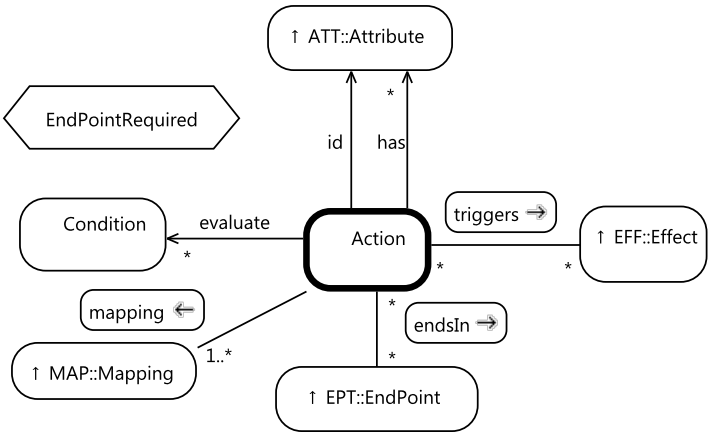


Figure 5.7: The `Actions'` structure diagram

given in Figure 5.9 and explained below.

`Action's` BD, Figure 5.8, shows two globally accessible operations, `CreateDefaultAction` and `AddMapping` as well as a constructor, `Default` for `Action`. The double-lined border shows that those operations modify and don't only query.

The local `EndPointRequired` invariant, Figure 5.9, expresses that an `Action` maps to either an `Effect` or an `EndPoint`. An analogous invariant on `Effect` verifies that there is indeed one `EndPoint`. The invariant forms a logical proposition that, in order to return true, must satisfy that there is either a tuple in the `trigger` relation or in the `endsIn` relation. This is expressed by comparing both relations to the empty set and negating that comparison.

The `AddMapping` operation, Figure 5.10, specifies the adding of a new

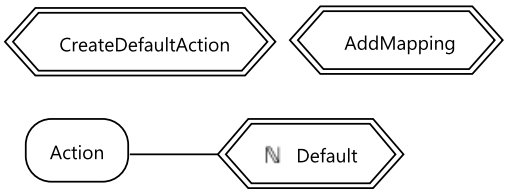


Figure 5.8: The `Actions'` behaviour diagram

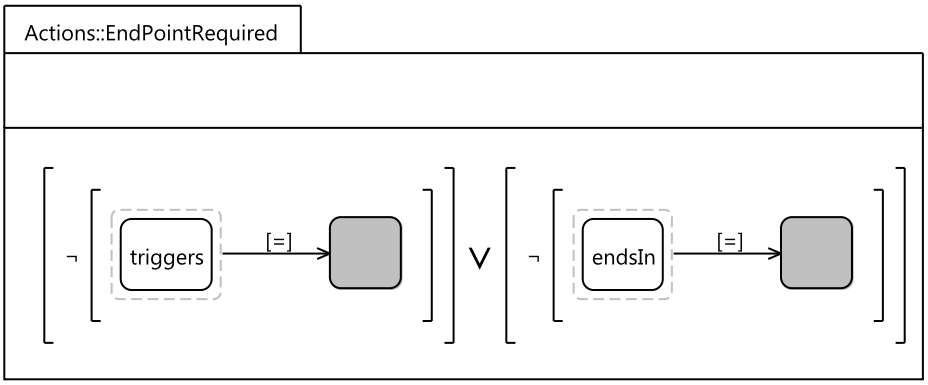


Figure 5.9: The `EndPointRequired` assertion expressing a local invariant on the `Action's` makeup

mapping by extending the set of tuples from `Action` to `Mapping` by one entry. To express this in VCL, the CD defines the new `mapping` relation to be the union of the old relation and the new tuple.

Attributes The *`Attributes`* package models the concept of `Attributes` as detailed in the specification of the widget structure. `Attributes` are used to describe physical and digital properties of a `Widget`. `Attributes` are key-value pairs. They are essential in the `Action-Effect-Endpoint Mapping` of the `Function` and as such, all `FunctionElements` have `Attributes` which are mapped from one element to the next. See Appendix A.3 for further details.

Effects The *`Effects`* package covers all concerns regarding `Effects`, which are an essential part of the widget behaviour. They specify what consequence a user or system action has. See Appendix A.4 for further details.

EndPoints The *`EndPoints`* package models part of the widget structure. Concrete `EndPoints` go unnoticed by users as they are only used by the

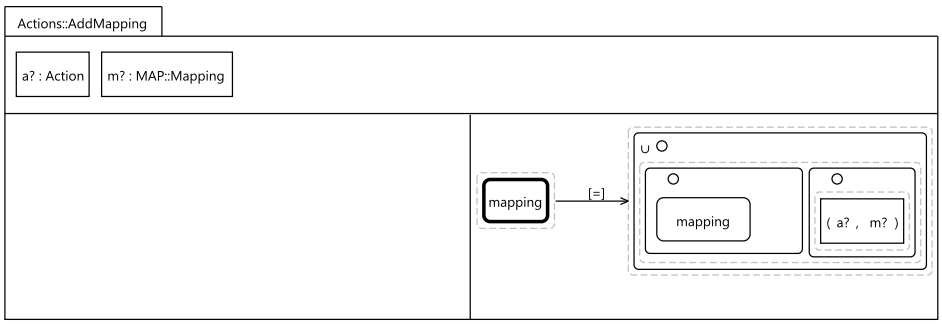


Figure 5.10: The *Action's* `AddMapping` operation

system to denote points of interaction and localise the `Effects` of `Actions`. See Appendix A.1 for further details.

Functions The *Functions* package models parts of the widget behaviour. `Functions` detail how user or system input, `Actions`, are linked to optional `Effects` and localised to `EndPoints`. A TWT user can specify concrete behaviour by mapping `Attributes` from one `FunctionElement` to the next. The behaviour is shown by a few token operations. Modelling the complete behaviour would have required a substantial amount of time and not added to the evidence needed to answer the research questions. Therefore, only a small part of the behaviour was specified. It is however sufficient to draw conclusions as the remainder of the behaviour would follow along the same principles, using the same primitives and syntax. See Appendix A.6 for further details.

Mappings The *Mappings* package laces `Attribute` together, specifying the input and output pipes for the propagation of `Attribute` values during a `Function`. See Appendix A.7 for further details.

Layers The *Layers* package models part of the application context as described by TWT. The behaviour is implicitly read from the specification. See Appendix A.8 for further details.

Widgets The *Widgets* package models the `Widget` and `Identity` concepts of the widget structure. `Widgets` carry behaviour as defined by their `Identities`. `Widgets` allow the user to transparently interact with TUI. See Appendix A.9 for further details.

TUI The `TUI` package is an ensemble and groups all concerns modelled by the other packages of the TUI metamodel in order to expose one top level

package, containing all concerns of the TUI domain specified by Bicheler. See Appendix A.10 for further details.

5.5 The BPMN2 VCL metamodel

This section discusses modelling choices and difficulties during the modelling of the BPMN2 specification [23] using VCL. All VCL packages can be found in Appendix B. As the specification is huge, some parts, mostly those not relevant to the modelling attempt, have been disregarded. Table 5.1, presented earlier, shows every concept that was discarded for the modelling. During the modelling, the specification was sometimes ambiguous and needed to be interpreted. These interpretations are made clear where necessary in the paragraphs below or the detailed description of the VCL packages in the appendix. In addition to the cuts from Table 5.1, all notions tying concerns from the table to modelled concerns have been cut, for example the concept of *Correlations*. Moreover, any recommendations, statements including "should" and "may" have been disregarded.

Due to the reduction of scope in regard to the original BPMN2 specification, the biggest change it the packaging scheme. While most packages have been retained as is, it was hard to identify clear cut packages in the first place. All efforts have been made to clearly separate concerns. Appendix B contains Table B.1 which shows all VCL packages and the corresponding diagrams. Figure 5.11 shows the *Core's* package diagram, giving the reader an idea about the structure of the BPMN2 metamodel by providing a view of the BPMN2's topmost package.

5.5.1 Packages

The following paragraphs detail all VCL packages for the BPMN2 metamodel using a bottom-up approach. However, due to the high coupling, a pure bottom-up approach was not possible

Primitives The *Primitives* container package defines all primitive sets used in the BPMN2 specification such as for example `String` or `Bool`. There is a no corresponding package in the BPMN2 specification as most of those primitives are built-in types. See Appendix B.1 for further details.

Extensibilities The *Extensibilities* VCL package includes all concepts necessary to allow for an extension of BPMN2 without sacrificing conformity. It aims at making BPMN2 an extensible language by adding the concept of extensibilities at a metamodel level, allowing any extended BPMN2 model to still be compliant with the specification of the core BPMN2. See Appendix B.2 for further details.

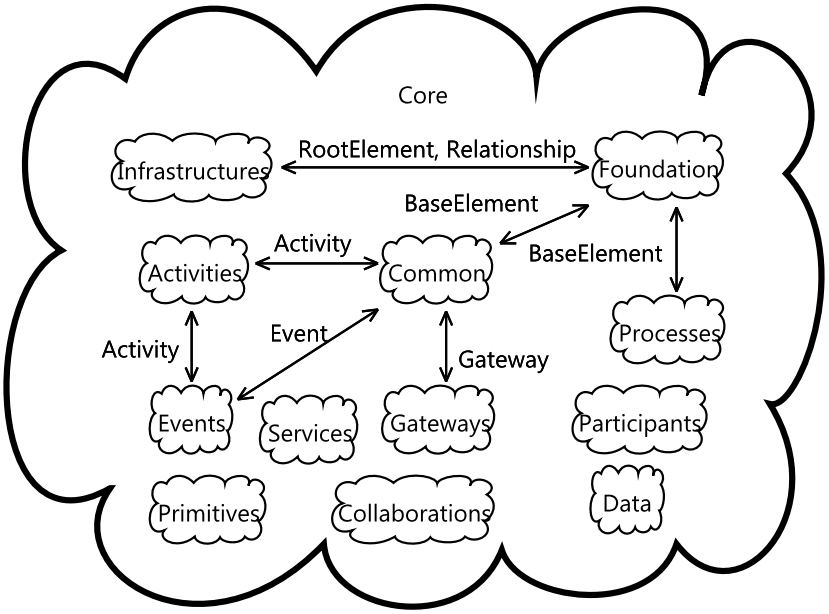


Figure 5.11: The *Core's* package diagram

BaseElements `BaseElement` is the topmost abstract class in BPMN2 for most model elements. The *BaseElements* package defines all subsets of `BaseElement`. In order to not violate the separation of concerns, the VCL package introduces the concept of `BaseElementContainer`. For further details, see Appendix B.3.

Foundation The *Foundation* VCL ensemble package groups all concerns from the *BaseElements* and *Extensibilities* VCL packages. It is modelled after the BPMN2 core structure which contains, amongst others, a BPMN2 *Foundation* package. See Appendix B.4 to consult all VCL diagrams.

Infrastructures The *Infrastructures* VCL package introduces two concepts, `Definitions` and `Import`. The first concept is the topmost concrete object for all BPMN2 elements, defining their namespace and scope. `Import` is used to import foreign BPMN2 or non-BPMN2 elements into the current BPMN2 model space. See Appendix B.5 for further details.

ItemDefinitions The concept of `ItemDefinitions` is used to separate the structural definitions from model elements that are exchanged during a `Process`, thereby decoupling definition from use. See Appendix B.6 for all VCL diagrams.

Messages The *`Messages`* VCL package implements the concept of information exchange between `Participants`. While constraints apply to the sending and receiving of `Messages` all those constraints are expressed on the corresponding `FlowElements`. A `Messages` references an `ItemDefinitions`, defining its payload and a textual description to make it easily identifiable. See Appendix B.7 for all VCL diagrams.

Artifacts This *`Artifacts`* VCL package defines all concept revolving around annotating and conceptually grouping diagram elements without influencing the model itself. The package contains the definition of the `Artifact` concept, grouping the specific artefacts like `TextAnnotations`, `Associations` and `Groups`. See Appendix B.8 for further details.

Resources The *`Resources`* package implements the concept of resources as commonly used in business contexts. A named `Resources` is defined by the many `ResourcesParameters` it may have. The parameters are typed by and named by `ItemDefinitions`. See Appendix B.9 for all VCL diagrams.

ResourceAssignments The *`Resourceassignments`* VCL package models the concept of `ResourceRoles`, `ResourceParameterBindings` and `ResourceAssignmentExpressions`. See Appendix B.10 for further details.

Expressions The *`Expressions`* VCL package offers facilities to specify `Expressions` in both, natural and a formal language. See Appendix B.11 for further details.

Errors `Errors` are raised when BPMN2 operations or `Activities` do not behave as expected. See Appendix B.12 for further details.

Collaborations `Collaborations` are the top level concept for modelling BPMN2 interactions in the VCL model. The *`Collaborations`* VCL package groups all concepts surrounding `Collaborations`. Business interactions between different business agents, called `Participants` are visualised as pools, a concept borrowed from flowcharts. These `Participants` can include `Processes` and interact by the means of `Messages` routed using `MessageFlows`. See Appendix B.13 for further details.

GlobalTasks The *GlobalTasks* VCL package models all concepts surrounding globally available Tasks. BPMN2 GlobalTasks are a specialised subset of Tasks. See Appendix B.14 for further details.

Services The *Services* VCL package groups all concepts revolving around *Services*, Interfaces, and Operations. Interfaces define a set of supported Operations, providing services on given EndPoints. For further details, see Appendix B.15.

ItemAwareElements The *ItemAwareElements* package defines what is, by the specification, BPMN2's concept of variables. See Appendix B.16 for further details.

IOSpecifications The *IOSpecifications* package models the BPMN2 InputOutputSpecification concept which aggregates all forms of DataInput and DataOutput, alone or in groups by InputSets respectively OutputSets. The IOSpecification is referenced by Activities and CallableElements to define their inputs and outputs. The InputOutputBinding is used when working with Services to bind an InputSet respectively an OutputSet to an Operation. See Appendix B.17 for further details.

DataAssociations The *DataAssociations* VCL package includes all concepts for relating data from one ItemAwareElement to the next. A DataAssociation is either a DataInputAssociation or a DataOutPutAssociation. See Appendix B.18 for further details.

Data The *Data* VCL package groups all concerns from the *ItemAwareElements*, *IOSpecifications* and *DataAssociations* VCL packages. There is no corresponding BPMN2 package although a *Data* is hinted at on multiple occasions, especially in the chapter on items and data. See Appendix B.19 for all VCL diagrams.

Processes The BPMN2 concept of Process is modelled in the *Processes* VCL package. The Process concept describes a sequence of FlowElements, Activities, Events and Gateways that are connected through SequenceFlows to define the execution semantics of a Process. See Appendix B.20 for further details.

Lanes The*Lanes* VCL package models the concept of BPMN2 lanes, the sub-divisions of pools. LaneSets are partitioning a Process to show individual roles within it. LaneSets can also be contained within Lanes, which have to be contained within LaneSets. This nesting allows to model complex business structures. See Appendix B.21 for further details.

Escalations An `Escalation` in a business situation is an exceptional situation during `Process` execution that the `Process` forwards to a higher instance to be handled. See Appendix B.22 for further details.

EventDefinitions The `EventDefinitions` VCL package defines all concepts surrounding the trigger of `CatchEvents` or outcomes of `ThrowEvents`. To account for the different causes, `EventDefinitions` contains individual subsets modelling each cause. See Appendix B.23 for further details.

Events The *`Events`* VCL package illustrates the concepts of `Events` in BPMN2. `Events` actively influence the flow of a `Process`. They either spring from a `Process` in the case of `ThrowEvents` or impact the current `Process` in the case of a `CatchEvent`. The `StartEvent` is the entry point of a `Process` and is a specialised `CatchEvent`. `IntermediateCatchEvents` and `IntermediateThrowEvents` can, as their name suggests, happen as the `Process` flow is executed. `IntermediateThrowEvents` serves to produce `Events` and resume normal process flow. `BoundaryEvents` are a specialisation of `CatchEvents`. They are always attached to an `Activity` and visualised by an event marker on the boundary of an event. They are used to catch abnormal interior flow, that is, erroneous behaviour, and offer a mean to deal with the consequences of the exception flow. `EndEvents` terminate the process flow. See Appendix B.24 for further details.

FlowElements The *`FlowElements`* VCL package groups all concepts from business process flows. All of these concepts can appear in `Process` flows. `FlowElements` is the overarching concept, encompassing more concrete concepts such as `FlowNodes`, `SequenceFlows`, `DataObjects` and `DataStoreReferences`. `SequenceFlows` are used to instil an order in the arrangement of other `FlowElements`. They map a source `FlowNode` to a target `FlowNode`. `FlowNode` is used as a concept to defining all elements that can be the source and target of `SequenceFlows`. See Appendix B.25 for further details.

FlowElementContainers The *`FlowElementContainers`* VCL package introduces a superset of container elements on BPMN2 diagrams. `FlowElementsContainer` is defined though two subsets, `Process` and `SubProcess`. These elements, by the `flowElements` relational edge, contain `FlowElements`. Moreover, Lanes can also be contained in `FlowElementsContainers`. See Appendix B.26 for all VCL diagrams.

CallableElements A `CallableElement` is a superset for all `Activities` that are not defined within a `Process` and can be reused by being referenced from within a `Process` or by a `CallActivity`. See Appendix B.27 for further details.

Common The *Common* VCL ensemble package logically group other packages. *Common* includes all packages that specify concepts used across any diagram types in BPMN2. It groups the following packages by enclosing them;

- the *ItemDefinitions* VCL package,
- the *Messages* VCL package,
- the *Artifacts* VCL package,
- the *Resources* VCL package,
- the *ResourceAssignments* VCL package,
- the *Expressions* VCL package,
- the *Errors* VCL package,
- the *FlowElements* VCL package,
- the *FlowElementContainers* VCL package,
- the *CallableElements* VCL package.

See Appendix B.28 for all VCL diagrams.

LoopCharacteristics `Activities` can be executed multiple times in a row, necessitating the concept of loops. The *LoopCharacteristics* VCL package models this concept. See Appendix B.29 for further details.

SubProcesses `SubProcesses` are `Activities` that model the internal process flow just as `Processes` do. The concept is modelled in the *SubProcesses* VCL package. `SubProcess` has two subsets, `Transaction`, modelling atomic `Activities` that specify a protocol to guarantee atomicity, and `AdHocSubProcess` which defines not one but multiple `Activities` for which the execution is dependent on the `Activity's Performer` although an `ordering` might impose some constraints on the order of execution. See Appendix B.30 for further details.

Activities Work in BPMN2 diagrams is represented by `Activities` which are part of `Processes`. Abnormal execution of an `Activity` is handled by `Events` attached to its boundary. See Appendix B.31 for further details.

Gateways The *Gateways* VCL package groups the concept of `SequenceFlow` control elements. A `Gateway` is defined in VCL by its subsets; `InclusiveGateway`, `ExclusiveGateway`, `ComplexGateway`, `EventBasedGateway`, and `ParallelGateway`. `Gateways` specifies a `GatewayDirection`.

An ExclusiveGateway and InclusiveGateway implement the concept of a branching point if the GatewayDirection is Diverging or of a merger if the GatewayDirection is Converging. A ParallelGateway is used to merge parallel SequenceFlows or to create them. ComplexGateways are used to model more complex flow decisions than are offered by the ExclusiveGateway. An EventBasedGateway is used as a branching point with the decision about the branching relying on an Event. See Appendix B.32 for further details.

Tasks Tasks are atomic Activities that users or the business application uses to handle work. As all Activities, they are contained within a Process' flow. A Task is defined by its seven subsets: BusinessRuleTask, ManualTask, ReceiveTask, SendTask, ServiceTask, ScriptTask, and UserTask. See Appendix B.33 for further details.

Participants The *Participants* VCL package includes all concepts revolving around interacting business entities, called Participants. They take on specific roles with whom they then engage in Processes. Other than the Participant, the package also defines a PartnerEntity and a PartnerRole. Both behave similar and the only difference is that a PartnerEntity is used to specify a general role such as a company whereas the PartnerRole is used for more specific Participant behaviour such as that of a buyer or seller. The Participant also specifies a ParticipantMultiplicity which needs to be specified if the scenario requires multiple Participant instances. Consult Appendix B.34 for further details.

Core The *Core* VCL ensemble package is an analogy to the BPMN2 core package, grouping all BPMN2 concepts in one package. More details can be found in Appendix B.35.

5.6 The Business-to-Table concept and metamodel

The Business-to-Table (BtT) metamodel is charged with mapping the user actions from the TUI onto the BPMN2 model. To this end, the model hooks into the ApplicationLayer of the TUI who has access to all Zones exposed to users. It will also define a set of BPMN2 metamodel elements it will directly be referring to as most of the elements are transparent for the users. This concept effectively builds a bridge, mapping TUI interactions with Zones onto BPMN2 model changes. The Business-To-Table concept only partially models the final application that will enable BPMN2 to be modelled on a TUI. The representation of the visuals of the model has been abstracted as it would have meant to design a bigger model which wouldn't have been more appropriate to answer the research question. Speaking in

the Model-View-Controler (MVC) design pattern, the "View" has not been modelled.

As by the TWT [104], `Actions` will produce `Effects`. These `Effects` will be creating, deleting, or manipulating model entities. The `Effects` will target `Zones`. Applications "listening" `Zone` will generate a packet wrapping the user interaction. These packets will be queued and processed, their manipulation in regard to the model decoded and categorised. User manipulations will affect only a limited number of model entities but those need to take care to properly change all related entities as well that may be indirectly affected.

The metamodel has been drafted using VCL. All VCL diagrams can be found in Appendix C. The metamodel has been split into two packages, each addressing one specific concern. Table C.1 lists all packages and the included VCL diagrams from the appendix.

Mutator The *`Mutator`* package is used to define all objects of BPMN2 which can be directly manipulated by user interaction. These elements are exposed as `Subjects` and are being addressed by the resolved user manipulations, the `Happenings`. This design decouples the definition of what can be manipulated from how it can be manipulated. For additional details and VCL diagrams, consult Appendix C.1.

BusinessToTable The *`BusinessToTable`* package models a bridge from user manipulations forwarded by `Zones`, defined by the TUI metamodel, to elements of the BPMN2 metamodel grouped as `Subjects` from the *`Mutator`* package. This package models the use of `Packets` to propagate the user interactions, their mapping to a specific `Happening`, and how those impact `Subjects`. Further details as well as all VCL diagrams are given in Appendix C.2.

5.7 The Ideation model

This section briefly describes the package that expressed the ideation model as described in Section 6 and displayed in Figure 6.1. The model has been designed using VCL. All VCL diagrams can be found in Appendix D. The appendix includes a list of all VCL packages and their diagrams in Table D.1. The model is an instance of the BPMN2 metamodel dressed earlier in Section 5.5. As VCL does not feature primitives to express this, the model does not enforce any checking on the correctness of the model instance.

Ideation The *Ideation* includes the complete model of the ideation scenario. The scenario is split amongst three lanes contained in one main pool. The modelling focuses on the `Ideator` lane which includes a process modelling the submission of an idea into a system to handle, propagate, and help refine the idea using peer reviews. Upon starting the process the user will be able to either submit an idea or participate in the elaboration of en existing idea. The process then handles the different activities to deal with the tasks at hand before, by looping onto itself, propose the initial choice again or termination of the process. See Appendix D.1 for further details regarding the implementation in VCL.

5.8 The Widget model

This section describes the widget model, the set of widgets used in the final prototype designed by Bicheler [104] for use in the ideation scenario as described in Chapter 6. The model defines four widgets for, zooming, stamping elements onto the canvas, annotating model elements, and linking these elements. All VCL diagrams can be found in Appendix E. The appendix includes a list of all VCL packages and their diagrams in Table E.1. The model is an instance of the TUI metamodel dressed earlier in Section A. As VCL does not feature primitives to express this, the model does not enforce any checking on the correctness of the model instance.

Due to the late date of the last prototype and the available document, the prototype model is only partially implemented. The problems required some drastic changes to the TUI metamodel which was needed to be able to specify this model.

Prototype The *Prototype* package models the four different widgets catering to the functionality listed above. The `Zoom`, `AnnoMarker`, `Link`, and `Stamp` widgets are held by the `WidgetLayer`, modelling the layer specified by the TUI metamodel that holds and manages all widgets. The model also specifies all visuals and handles attributed to the individual widgets. See Appendix E.1 for further details regarding the implementation in VCL.

The four different behaviour packages, *ZoomBehaviour*, *StampBehaviour*, *LinkBehaviour*, and *AnnotationBehaviour* were designed to separate the concerns regarding the identities of the widgets, expressing their behaviour, from the concept of widgets and handles modelled by the *Prototype* package. Due to problems listed above, only the *ZoomBehaviour* package was implemented. The package defines the `ZoomBahaviour` concept and the three functions that specify the behaviour; `DropBind`, `LiftUnbind`, and `RotateZoom`. See Appendix E.2 for further details regarding the implementation in

VCL.

5.9 The Interaction model

Unfortunately, due to the need to rework some of the previous models with last minute changes, the interaction model was not implemented.

5.10 Conclusion

This chapter answers the third research question through answering three related sub-questions;

RQ3 *Is it realistic to use a GPVML to model complex applications?*

- **RQ3a** *Can concerns be separated in a meaningful way?* — **Yes**
- **RQ3b** *Can all requirements be modelled?* — **No**
- **RQ3c** *Is the expressiveness of VCL sufficient?* — **No**

The following sections address each of the three sub-questions raised above and then gather all these answers and formulate an answer to the main research question.

5.10.1 Separation of Concerns

VCL uses a packaging scheme that divides concepts into packages. The Visual Contract Builder (VCB), VCL's tool, assists the packaging scheme by providing ready to use VCL package. However, all packages lie on the same level. For example, the concepts of TUI and BPMN2 are clearly distinct. Yet all TUI and BPMN2 packages reside on the same level in the tool. As it does not change the actual separation of concerns, it has no impact on the models but merely on the ease and comfort of modelling.

Nevertheless, the packaging scheme allows to separate concerns neatly while the diagrams contained in each package enable the modeller to separate the structure from the behaviour and the constraints. The distinction between local and global scopes makes it possible to expose operations from a package while keeping others hidden, enabling the modeller to use encapsulation and all its benefits.

While some improvements can be made on how the tool presents packages, the overall packaging scheme allows for a meaningful separation of concerns and the encapsulation of local operations. Therefore, research question RQ3a can be answered with a definite "yes".

5.10.2 Requirements coverage

During the modelling of the different metamodels and models it has become clear that not all requirements could be modelled. The TUI metamodel was designed encountering only one invariant that could not be formulated. It was only when dealing with the more complex BMPN2 specifications that more difficulties arose. The remaining metamodel and models were again drafted without any problems due to the low number of constraints. This leads to the observation that the other specifications may have been incomplete or under-specified. This confirms the necessity and validity of using the specification a well established and specified notation during the modelling process. All shortcomings of VCL in regards to the coverage of requirements that were discovered during the modelling process are summarized in the following two paragraphs. These take only the parts of VCL into account that are offered by the VCB.

Lack of quantifiers During the formulation of invariants for the BPMN2 metamodel, many of the invariants failed to correctly express the constraints. This is due to them being expressed in natural language using absolutes such as "no", "all", or "each". In first order logic, these translate to universal or existential quantifiers. These do unfortunately not exist in VCL and, therefore, expressing these invariants is more complicated. An example of how universal invariants can be expressed using VCL can be found in the `FlowElementContainers' ExecutedImmediately` invariant found in Section B.26 of Appendix B. For the reader's convenience, the invariant is reproduced in Figure 5.12.

The invariant formulates a predicate expressed that the `flowElements` set containing tuples of the kind (`FlowElementsContainer`, `FlowElement`) must be empty. The range of the set is restricted to executable `Processes` and given a range of non-immediate `SequenceFlows`. This invariant expressed that there cannot be such a tuple less the invariant fail to hold. Similarly, not shading the predicate, expresses that such as set must exist. This strategy works sufficiently well for absolutes on sets.

Unfortunately, the same strategy cannot be applied when specific set elements need to be picked and matched. For example, the `FlowElements' MessageFlowsSpanPools` invariant fails to meet its purpose. What the invariant should have expressed is that no two `Participants` or `FlowNodes` connected by a `MessageFlow` can reside in the same `Collaboration`. The invariant is in need of selecting two distinct elements from a set. However, in VCL, assertions do not allow to define variables less quantify. Hence, the impossibility to express the invariant.

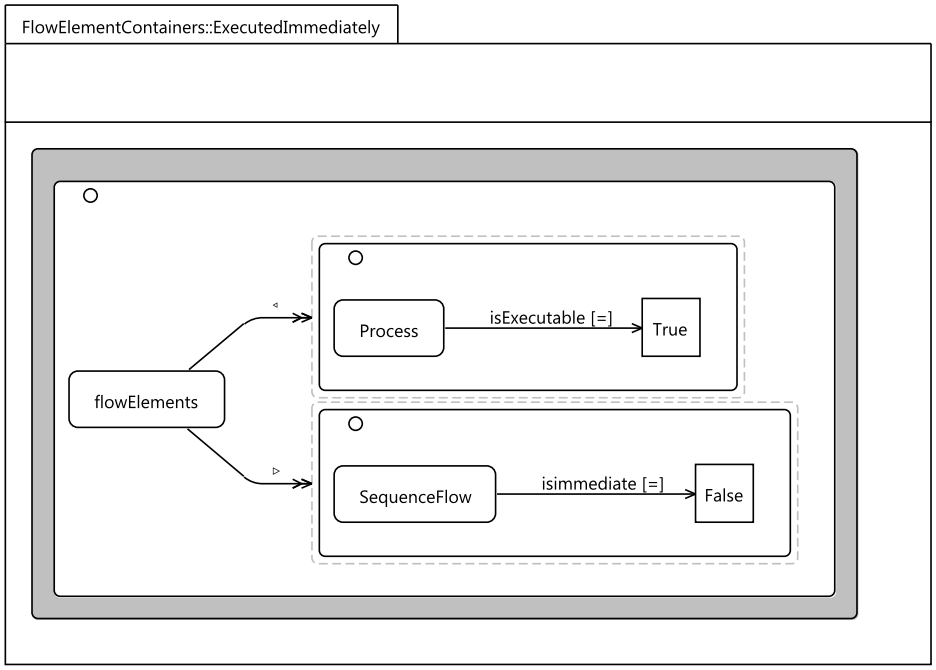


Figure 5.12: The `ExecutedImmediately` invariant expressing constraints on `Processes` execution

Transitive Closure The TUI metamodel specification mentions the need of a `Function` to start with an `Action`, possibly passing by one or many `Effects`, before finishing in an `EndPoint`. This is only partially modelled in the structural diagram of the *`Function`* package found in Section A.6 of Appendix A. `Actions` specify the mapping of `Attributes` to `Effects` and those need to express the mapping towards, once the possible `Effect` loop finishes, an `EndPoint`. The `EndPointRequired` invariant fails to express this constraint. It only models that an `Effect` maps to either an `EndPoint` or another `Effect`. A model instance could be produced where the `Function` contains an arbitrary `EndPoint` not linked to the `Effect`, with the latter only mapping to itself. While this would satisfy the metamodel, it would be a counter example, showing the requirements have not been met.

Therefore, what is required is the ability to express a transitive closure in order to formulate an invariant on `Action` that either must directly map to an `EndPoint` or that by the transitive closure of the mapping towards `Effect` is mapped to an `EndPoint`. This would satisfy the requirements of the TUI specification. As not all requirements could be modelled, research question RQ3b can be answered with a definitive "no". However, it is noteworthy

that the team is aware of some of the deficiencies and working on finding a way to implement them. At this point, quantification is one of the next points on the agenda.

5.10.3 Expressiveness

The modelling process was thought to model in a short time interval the modelling process of a bigger project to judge whether VCL does provide all the tools necessary to get the job done. As stated in the previous section, the language is lacking some syntax to suit the needs. Semantically the concepts used by VCL are sometimes a bit cumbersome. The use of sets is well suited to formally express ideas but is, unfortunately, not how people are used to view things. However, this should not be viewed as a drawback. The semantics are, subjectively, easier to understand than formal mathematical statements.

Nevertheless, some features are missing. VCL does not provide a facility describing properties of both, models and metamodels. It is not possible to model, for example, that a package defines a model that is an instance of a metamodel. Given the modelling scenario at hand, the requirement to model both, models and metamodels, and that VCL is not providing said facilities, RQ3c cannot be affirmed.

As was the case previously, the VCL team is aware of this particular lack in expressiveness and is looking into adding the concept. Some features were added since the start of the thesis, addressing the lack of expressiveness regarding the set cardinality, the # operator, and the retrieval of optional properties by the unary ($\odot$) operator. Unfortunately, the unary operator only works on optional properties, those marked by an ? cardinality. A selector operator would be required to select a property in a non-local scope. However, VCL does not offer said selection operator. Moreover, the use of the recently introduced cardinality operator is restricted to a few places and could benefit from opening up the places where it can be used. An example would be the `SubProcess' OneStartEvent` invariant, which would need to restrict the domain with a predicate on the cardinality.

5.10.4 Conclusion

Modelling a complex application requires a lot of effort and, if the modelling is done formally, a mature language offering syntax and semantics able to address all needs is required. With only one out of three sub-questions having been affirmed, VCL does not offer all the tools necessary to model the scenario at hand. Yet, the study in Chapter 3 showed that VCL was subjectively to be the best of the available GPVML. Looking at the other

languages that were examined, it is unlikely that any would have outperformed VCL. This seemingly leads to the conclusion that no GPVML is, as of the time of writing, mature enough to be used in modelling complex applications. However, research question RQ3 asks;

RQ3 *Is it realistic to use a GPVML to model complex applications?*

The modelling that had been done during the thesis showed that it was indeed realistic to model complex applications using GPVML if realistic is to be interpreted in the light of feasible. Nevertheless, the goal of using a formal VML would have been to formally define the application. This was not possible due to the problems stated in the previous sections.

However, with VCL still being in development, it is not farfetched to assume that investigations such as this serve to highlight deficiencies which are then addressed by the team. Moreover, the parts of VCL that are used by the VCB are only a subset of VCL. Quantification for example does exist in VCL, it just hasn't been formulated for VCB yet. If the shortcomings that VCL has are fixed in the future, it could very well be one of the first formal GPVML that can be used to model complex applications.

Chapter 6

Ideation

This study is designed to answer the first research question found in Section 1.1;

RQ1 *Is it possible and practical to model an application or process using a TUI?*

To find an answer, a scenario was created using ideation, a research topic of the Public Research Centre Henri Tudor. The goal of the project is to investigate and formulate a business process for submitting ideas, sharing them, asking for them to be peer-reviewed, and participate in help devising other co-worker's ideas. This chapter details a small part of the ideation process as most of it is confidential. The ideation process is modelled using the Business Process Model Notation 2 (BMPN2) on a Tangible User Interface (TUI). The TUI used for the evaluation is the tabletop based TUI presented in Section 5.2.1. A VCL model of the ideation scenario can be found in Appendix D.

The scenario was split into small steps and resumed in a companion document that was distributed to a set of participants. Three sessions were held together with Bicheler, supervisors from the Public Research Centre Henri Tudor, and the test candidates. The feedback from the participants was recorded in order to evaluate the test performance. Section 6.1 will introduce the scenarios for the test sessions as well as the technical aspects of the study. Sections 6.2, 6.3, and 6.4 will detail the circumstances in which each of the three tests was run and give the feedback gathered from the test candidates. Section 6.5 will present an analysis and evaluation of the results from the test sessions. Finally, Section 6.6 wraps up the study, resumes all findings and gives an answer to the research question.

6.1 Introducing the scenario

In the course of this scenario the participants of the tests will use widgets. Widgets are tools (Zoom, Stamp, Chain, Link Components and Annotation Marker) composed of one or more physical handles and appropriate graphical representation on the table surface. Physical handles are small devices in the shape of, in this case, Sifteo cubes, that help the user to manipulate the business model on a tabletop TUI. The Zoom widget was to be used like a radial. Stamp and Chain widgets are used as they were designed for the case study. Their description can be found in Table 3.1, The linking of components is done by using a Link widget with two physical handles that are attached to the components connecting to the head and tail of the arrow. Annotations were added using a multi-handle widget. One handle selected the component to be annotated while a second handle, a Wacom Intuos4 PTK-540-WL Wireless Tablet was used to capture the writing of the test candidates.

The testers participated in three test sessions. All session were played with two candidates each. Each session featured the same basic ideation scenario. A companion document details the scenarios step by step and explains what widget to use and how to use it to fulfil a step. A simplified version of said document detailing all tasks of the first test can be found in Appendix H.

Figure 6.1 shows the final BPMN2 model to be achieved after all steps have been fulfilled. The corresponding VCL model and all related diagrams can be consulted in Appendix D. In each step, the participants will be asked to add one or more model entities, building the final model from scratch. In addition to the model components, a separate area known as toolbox is present on the surface. This toolbox is partitioned into two sub-areas: one contains rectangles with the name of widgets, the other contains BPMN2 model entities. The palette with model entities is intentional. However, the toolbox containing widgets is a workaround to create widgets which, ideally, should have been created before with fixed functionality. This was not possible due to the Tangible Widget Toolkit (TWT) still being in development.

6.1.1 Extending the model

The second test scenario was build upon the first scenario. It can be seen in Figure 6.2. Components were added to the model that were impossible to model with the current set of widgets. The goal was to observe how the participants reacted and how they interacted with the TUI, using what widgets, to attempt to solve the problem. Moreover, their feedback regarding that

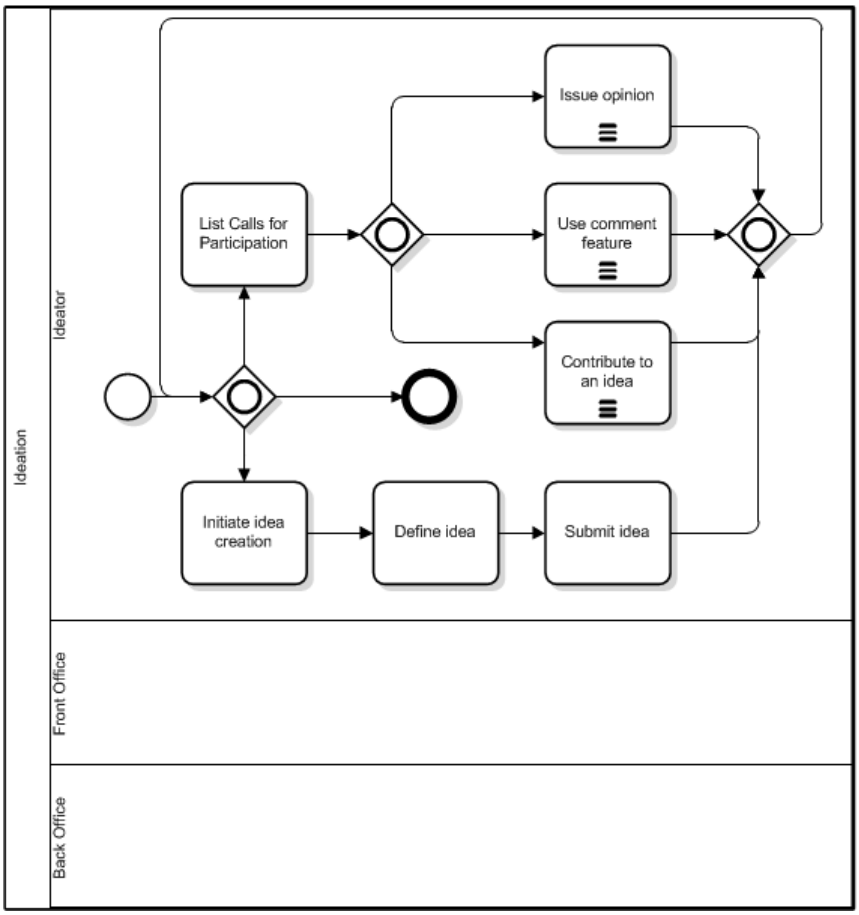


Figure 6.1: The ideation scenario in BPMN2

situation was used to validate the conception of more widgets. The scenario also served to confirm results obtained from the first, crude prototype used in the first test.

6.2 First test scenario

The first test session was played with a very early design of the widgets and no real prototype. The session was played using paper mock-ups for model elements as well as visual feedback from the widgets. The Sifteo cubes were used to give the users a feel for how the widgets are going to feel later on. The feedback gathered from the first session is gathered below:

- **Need of feedback**
  Firstly, the candidates mentioned the importance of dynamic feedback for the placement of components. Secondly, the need of feedback re-

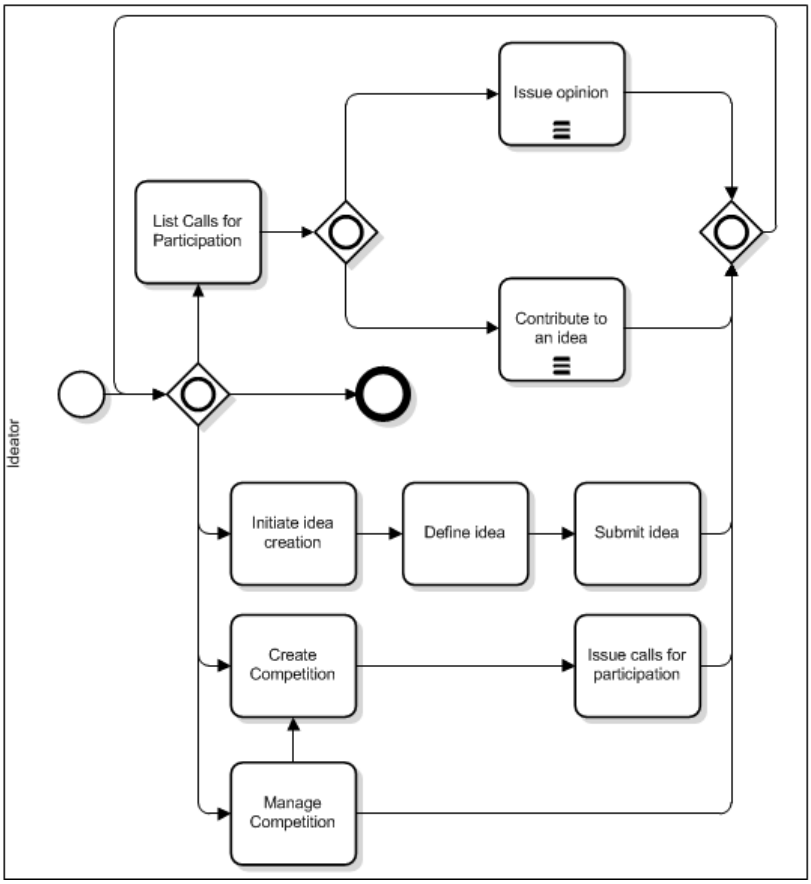


Figure 6.2: The ideation scenario extension

garding user choice as to increase usability. User selections should be displayed or confirmed in some way.

- **Rearrangement of components**
  The candidates felt that it was important and unavoidable to provide functionality to rearrange model components.

- **Use of templates**
  The candidates expressed the wish for facilities that would propose templates or commonly used "bulk" components to speed up the modelling process.

- **Table orientation**
  The candidates recognised that the environment is supposed to be collaborative and spotted the problem of table orientation. The problem is solved by a specialised widget to change table orientation.

- **Labelling**

The question was raised on how components are labelled as in this test scenario, labelling of components was not included.

- **Reaction time**
  The candidates expressed the need for a highly responsive system.

6.3 Second test scenario

The second test session was held with widgets realised with an early prototype of TWT. Therefore, the Chain widget was not implemented. Figure 6.3 shows the modelling process. The prototype widgets are crude, mainly due to the table's camera resolution which does not allow for the size of the fiducial markers to be small enough to become ubiquitous. Still, the participants were eager to explore the system, discovering the functionality of most widgets by themselves. The feedback of the second session is resumed below:

- **Need of feedback**
  The system still being an early prototype, no system feedback was given. This piece of user feedback strengthened the user's stance on it being a necessity to have feedback. User selections should be displayed or confirmed in some way.

- **Use of templates**
  In addition to the feedback given in the first session, participants also expressed that the user should be able to express what components to use as templates.

- **Labelling**
  Labelling proved useful although better feedback needs to be provided as to what component is chosen to be labelled.

- **Handle size**
  The testers noticed the handle being represented by two disjoint, physical objects, the marker and the cube, and noted that the mismatch in size was a bit irritating.

- **Persistence**
  The users expressed the wish to save and carry the model.

- **Model mutability**
  The participants expressed the wish to use domain models outside of their domain, providing the ability to overlay information from different domains.

Figure 6.3: Modelling during the second test

6.4 Final test scenario

The third and final test session was held using the final prototype developed by Bicheler in the scope of his Master's Thesis [104]. In addition to the widgets proposed for the second test phase, the Sword widget was implemented. The widget allowed to remove model components by applying the metaphor of *slashing* or *cutting*. Furthermore, the prototype provided visual feedback to the users, using the Sifteo's screen to show the identity of each widget. The addition of the Sword widget made it possible to fully play the extended test scenario shown in Figure 6.2. The final feedback is listed below:

- **Need of feedback**
  The implemented feedback provided useful. However, testers were observed to produce sound along with their interactions which leads to the conclusion that audible feedback is anticipated and therefore wanted.

- **Editing labels**
  Spelling mistakes made by the testers and their wish to correct them led to the assumption that a label edit functionality is needed.

- **Rearranging elements**
  Testers expressed the wish to rearrange model elements.

- **Fluidity**
  Due to the hardware limitations of the tabletop TUI, the fluidity of

the modelling process was hampered. The test group saw this as inconvenience.

- **Canvas space**
  Test participants expressed the need for a bigger modelling surface.

6.5 Analysis & Evaluation

During all tests, none of the users ever felt discouraged or significantly hampered by the interface. All of them acknowledged that the hardware limitations were possible to overcome with future implementations of the table, that there was room for improvement. Therefore, none deemed the interface unusable. Judging their interactions with the interface, the other tester on the table and the observers, it has become clear that they were having fun exploring and using the interface. While this was not the goal, it shows that the participants felt at ease and not stressed even though one test scenario was held with a tight time frame in mind.

Most of the feedback was anticipated. The need for haptic, visual or auditory feedback from the system was partially addressed with the completion of the prototype. Having feedback did indeed improve usability and participants, now that they were able to clearly identify the widgets and their states. The user making audible explosion like noise upon deletion of a model component shows that the system would benefit from audible feedback as some users seem to expect it.

The testers discovered by themselves that the scenarios were intended to be modelled cooperatively. They naturally shared tasks and each took on a "role", catering to a group of widgets exclusively. There was a lot of discussion and information was shared efficiently, never being misunderstood or misinterpreted. It seems that either the scenario was to simple or that the TUI helped a lot to avoid misconceptions. Of course, the small test group did also play a role to avoid confusion.

From discussions during and after the test sessions, the user reacted positively to the presentation of more widgets. When informed about future widgets like the Lasso, which allows grouping several zones by encircling them with a handle, all test subjects reacted positively and were able to suggest uses for the widgets. This confirms that the metaphors that were chosen for the widgets designed in collaboration with Bicheler. All widgets, including those not used in the test scenarios that were conceived can be found in Tabled 5.1 to 5.3 of Bicheler's thesis or in Appendix J.

The users expressed ideas and concepts that were not captured by the test scenario. This leads to two observations. Firstly, as the tester's familiarity with the TUI grows, they are better able to comprehend the metaphor and apply it to new domains. Secondly, user behaviour in TUI can be rather unpredictable. This has been anticipated and the need for an automated verification of the modelled elements against a metamodel in order to provide feedback about the diagram validity is a necessity. Unfortunately, at this time, integrating such provers that, ideally, operate in real time is a rather impossible task. Mainly because validation in real time is not given and secondly because metamodels are not always available, at least not in a way that would enable provers to use them.

In post-session talks, the testers were taken with the possibilities the TUI provided. During the discussion it became clear that they did indeed see potential in the use of TUI. They expressed the wish to use previously modelled domain-specific models as a basis to model other domains on top. For example, a business model could be enhanced by modelling the distribution of resources, overlaying two models. Also, the possibility to use widgets to carry models from table to table or record the modelling process for it to be replayed and better analyse critical design decisions were discussed.

During the evaluation talks at the end of each session, especially the last ones, it became apparent that the features requested by users for TUI could benefit from other novel approaches. For example, the combination of different models, overlaying one onto the other can be realised with current file systems. However, TUI could benefit from a Snippet System [117, 118]. The system proposes to add pieces of files and from the pieces form a whole rather than have one file. The benefit is that content is not duplicated but referenced. Hence, models can be refined by specialists in non-collaborative scenario and still be up to date when resuming collaborative modelling attempts without the need for versioning systems.

6.6 Conclusion

In order to answer the first research question, a study was designed and executed with a small sample group. The question captures feasibility and practicability of a TUI modelling approach. After the study, the feasibility has been demonstrated and that part of the question can be answered with a definitive "yes". The practicality is harder to judge. Observing the test groups, it was clear that they were working collaboratively, exchanging ideas quickly and, without an extended learning phase, were able to model the scenarios. Without any drawbacks other than those imposed by the hardware or the prototype software, it is safe to say that it is not impractical.

While this does not answer the research question, it does not lead to a negative conclusion, indicating that further studies are necessary.

The interface allows for a collaborative modelling activity without the testers feeling as if they faced a bigger overhead or lost time in regard to traditional single-user modelling. Time-wise, the focussed, non-collaborative modelling attempt may be faster, however, no measures were collected in that regard. Nevertheless, the additional information conveyed in a collaborative scenario are thought to outweigh the possible time loss. Moreover, spoken information can be captured and processed to provide a sort of documentation that tracks the models evolution and design decisions. This is hardly possible in a non-collaborative environment. Therefore, the question about the modelling activity's practicality cannot definitively be answered as no hard measures are available. However, judging from the tests and the test group's responses, the question about the practicality could be "yes" but further studies will be needed to find a definitive answer.

The results encourage the research in the use of TUI for modelling activities. With the evolution of the interface, a larger study should be drafted to gather insights about modelling in larger groups of up to five people. Also, clear measurements regarding time and performance should be collected and compared to results obtained from modelling in a non-collaborative TUI environment as well as a standard editor-based single-user modelling environment. Ideally, future tests could also involve the proposition of novel TUI based applications and their acceptance. The tester's have expressed the wish for technologies such as the Snippet System, hence, further investigations could prove fruitful.

Chapter 7

Conclusion

This thesis studies VMLs and their applicability to TUIs and modelling TUI applications. Chapter 2 gives some background, exploring current research in all fields related to the thesis; modelling, visual modelling languages (VML), and tangible user interfaces (TUI). Chapter 3 elaborates on the design of a study and its measurements to explore the state of the art in VML. The results of the study point to the Visual Contract Language (VCL), detailed in Chapter 4, being the best suited VML to use in the modelling of a complex application based on TUI conducted in Chapter 5. And finally, the premise that TUI can be used to model applications and processes is investigated in Chapter 6.

The investigations conducted in these chapters serve to answer the research questions drafted in the introduction:

- **RQ1** *Is it possible and practical to model an application or process using a TUI?*
- **RQ2** *Which General-purpose VML (GPVML) performs best in modelling a VML scenario for use on TUI?* — **VCL**
- **RQ3** *Is it realistic to use a GPVML to model complex TUI applications?*

The thesis starts by tackling research question RQ2. Chapter 3 conducts a study to identify available VMLs, which are subject to selection criteria designed to select suitable VMLs. Three languages, Augmented Constraint Diagrams (ACDs), VCL, and UML in unison with the Visual Object Constraint Language (VOCL) came out on top. The languages are compared using a TUI scenario.

The measurements collected during the study include; tool support, semantics and language transformations, expressiveness, usability, error check-

ing capabilities, and model verification facilities. These measures were weighted by responses from a small questionnaire and tallied to present the results. These showed that VCL was able to get the highest score, mostly because the language is still being actively supported and offers a rich modelling tool, the Visual Contract Builder (VCB). The results of the study are published in a technical report at the University of Luxembourg's LASSY [70]. They have also been submitted and accepted as an extended abstract to the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) [119].

Having determined what VML to use, Chapter 4 gives a brief introduction to VCL. It is then used in a more elaborate scenario to answer research question RQ1. The scenario used in Chapter 5 is a result of the collaborative work and synergies from Bicheler's Master's Thesis [104]. His thesis lays the foundations for a TUI Widget Toolkit (TWT) used in conjunction with BPMN2 to define a BPMN2 business process, ideation, in a TUI setting. The complexity of the modelling endeavour stems from the need to model metamodels, models, as well as all constraints for each individual application element. This scenario proposed three narrower research questions:

- **RQ3a** *Can concerns be separated in a meaningful way?* — **Yes**
- **RQ3b** *Can all requirements be modelled?* — **No**
- **RQ3c** *Is the expressiveness of VCL sufficient?* — **No**

The only question that could be satisfied was RQ3a due to VCL and VCB providing sufficient and neat packaging and encapsulation schemes. Due to, among others, the lack of support for quantifiers in VCB, they are, however, present in the core definition of VCL, not all requirements could be modelled. Therefore, RQ3b could not be affirmed. The same holds true for RQ3c as VCL currently lacks the necessary functionality to express metamodel instantiation.

These results are however not to be taken at face value. The Model Driven Engineering field and community are far from well established. There is still a lot of research going on and there does, as Chapter 2 suggests, no perfect solution. Looking at the development of VCL and VCB, a clear trend forward can be seen. VCL has already shown that it is capable of tackling challenges from the modelling and formal methods communities [120, 121, 122, 123]. Those studies were conducted by people working on VCL and VCB which meant that they could immediately expand and improve VCL as problems crept up. Despite all progress and development, the language and its tool are still not at a level of maturity to enable its general applicability in a wide variety of settings as has been shown in this

thesis. However, with the willingness to invest into the language and the tool that has been shown by the team, they will continue to improve and become more mature in the future. The question as to when VCL will be mature enough to enable its general applicability; will have to go unanswered. A good approximation, although only jokingly, would be along the lines of what Mellor always said [124]; three years.

Chapter 6 details the scenario used in the modelling process used to gauge the performance of VCL in regard to the previously mentioned research question. However, the chapter does not content with just giving the scenario. It was designed in collaboration with Bicheler to serve two purposes, serve as a basis to develop a TWT prototype for his thesis, and provide the basis for a study on investigating the feasibility and practicality of using TUI to model an application or process, which is covering RQ1. The study concludes that is definitely feasible to model using TUI but did not collect any hard measures regarding the practicality.

The introduction formulated the hypothesis that formal collaborative modelling environments, using visual notations to ease the modelling process for non-experts, may prove useful to remedy some of the deficiencies in the software industry. Unfortunately, it could neither be verified nor falsified during the thesis. However, some interesting findings have been made. The findings from Chapter 6 lends credibility to the parts of the hypothesis that stipulate modelling novices would benefit from visuals and the metaphors provided by the TUI. The chapter also showed that the benefits of the collaborative environment can be translated into the modelling process. However, as no hard measures were taken, a larger study is needed to settle the matter and get a definitive, quantitative measure.

With the current state of the art in modelling, it does however seem that visual modelling languages are not yet providing enough functionality or ease of use to provide all tools necessary to model a vast range of complex problems. VCL, however, has shown to be able to solve some complex modelling challenges listed previously; a study could be conducted that investigates the modelling of such a scenario on a TUI, bringing domain experts and engineers together. Such a study, if repeated and conducted properly, could lend more credibility to the hypothesis and serve as a proof of concept.

7.1 Future Work

The previous section already stated the need for studies collecting hard measurements. These studies would need to be conducted on a medium to large scale to provide enough data and objective test samples to lead to useful and valid conclusions. The studies could investigate, for example:

- the effects of cooperative modelling processes on reducing stakeholders' unrealistic expectations,
- the availability of domain knowledge in a cooperative modelling environment,
- the improvement of accessibility of the modelling process using visual modelling languages,
- the effects of metaphors to understand visual modelling languages,
- the ease of forming logical predicates using metaphors and visual modelling languages,
- the use of VCL and VCB in a cooperative, TUI modelling environment.

This thesis leads to the discovery of interesting TUI uses. For example, during the ideation scenario, testers mentioned that they would want to use models that they had just drafted and interleave them with models of different domains, using one in the context of the other. A concrete example would be to use a company's business model to model the distribution and affection of different resources. One could imagine that changes in one model could imply changes in the other model, either automated, accounting for any change in a predefined manner, or notifying an administrator of a change in the underlying model.

Moreover, a timeline functionality can be used to track model evolution on TUI, enabling new individuals to engage in the collaborative modelling process to quickly catch up with the process. Additionally, the TUI environment can be enriched with audio information or gesture recognition, captured and processed by the TUI for knowledge elicitation and inference with the goal of building richer and more comprehensible models.

Bibliography

[1] Frederick P. Brooks, Jr. No Silver Bullet – Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, April 1987.

[2] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[3] David Harel. Biting the silver bullet: Toward a brighter future for system development. *Computer*, 25(1):8–20, January 1992.

[4] David Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, May 1988.

[5] Allen Newell. *Human Problem Solving*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[6] Allen Newell and Herbert A. Simon. Computer Simulation of Human Thinking. *Science*, 134(3495):2011–2017, December 1961.

[7] Robert Oerter. *The Theory of Almost Everything : The Standard Model, the Unsung Triumph of Modern Physics*. Pi Press, July 2005.

[8] David J. Barnes and Dominique Chu. *Introduction to Modeling for Biosciences*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[9] A. Henderson-Sellers and P. J. Robinson. *Contemporary climatology*. Longman Scientific & Technical, Harlow, Essex, 1986.

[10] Marleen H.F. McCardle-Keurentjes, Etinne A.J.A. Rouwette, and Jac A.M. Vennix. Effectiveness of group model building in discovering hidden profiles in strategic decision-making. In *Proceedings of the 26th International Conference of the System Dynamics Society*, pages 1–13, Albany, NY, USA, 2008. System Dynamics Society.

[11] Peter W. Higgs. Broken symmetries and the masses of gauge bosons. *Physical Review Letters*, 13:508–509, October 1964.

[12] The Standish Group. CHAOS Report. Technical report, The Standish Group, 1995.

[13] Diane Crawford. Forum: software project failure lessons learned. *Commun. ACM*, 42(11):21–24, November 1999.

[14] The Standish Group. CHAOS Summary 2009. Technical report, The Standish Group, 2009.

[15] Maria F. Costabile, Daniela Fogli, Catherine Letondal, Piero Mussio, and Antonio Piccinno. Domain-Expert Users and their Needs of Software Development. In *UAHCI Conference*, pages 232–236, Crete, June 2003.

[16] P Goolkasian. Pictures, words, and sounds: from which format are we best able to reason? *The Journal of General Psychology*, 127(4):439–459, October 2000.

[17] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, 1987.

[18] D Moody. The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *Software Engineering, IEEE Transactions on*, 35(6):756–779, 2009.

[19] Orit Shaer and Eva Hornecker. Tangible user interfaces: Past, present, and future directions. *Foundations and trends in Human Computer Interaction*, 3(1–2):1–137, 2009.

[20] Valérie Maquil, Eric Ras, and Olivier Zephir. Understanding the characteristics of metaphors in tangible user interfaces. In *Workshop Proceedings of Mensch & Computer*, Chemnitz, Germany, 2011.

[21] F Boers. Metaphor awareness and vocabulary retention. *Applied Linguistics*, 21(4):553–571, December 2000.

[22] Roy Schmidt, Kalle Lyytinen, Mark Keil, and Paul Cule. Identifying software project risks: An international delphi study. *Journal of Management Information Systems*, 17(4):5–36, March 2001.

[23] Object Management Group. Business Process Model and Notation. Specification, Version 2.0, Object Management Group, 2011.

[24] Robert Baillargeon, Robert France, Steffen Zschaler, Bernhard Rumpe, Steven Völkel, and Geri Georg. Workshop on modeling in software engineering at icse 2009. *SIGSOFT Software Engineering Notes*, 34(4):34–37, July 2009.

[25] Steve Easterbrook and John Callahan. Formal methods for verification and validation of partial specifications: A case study. *Journal of Systems and Software*, 40(3):199–210, March 1998.

[26] Daniel Jackson and Martin Rinard. Software analysis: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 133–145, New York, NY, USA, 2000. ACM.

[27] Alessandro Cimatti, Fausto Giunchiglia, Giorgio Mongardi, Dario Romano, Fernando Torielli, and Paolo Traverso. Model checking safety critical software with spin: An application to a railway interlocking system. In *Proceedings of the 17th International Conference on Computer Safety, Reliability and Security*, SAFECOMP '98, pages 284–295, London, UK, UK, 1998. Springer-Verlag.

[28] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Communications of the ACM*, 53(2):58–64, February 2010.

[29] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[30] Rajwinder Kaur Panesar-Walawege, Mehrdad Sabetzadeh, and Lionel Briand. Using model-driven engineering for managing safety evidence: Challenges, vision and experience. *Software Certification, International Workshop on*, 0:7–12, 2011.

[31] M. Chen, J. F. Nunamaker, Jr., and E. S. Weber. Computer-aided software engineering: present status and future directions. *SIGMIS Database*, 20(1):7–13, April 1989.

[32] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006.

[33] The Eclipse Foundation. Eclipse modelling project. `http://www.eclipse.org/modeling/`. [Online; accessed 17$^{\text{th}}$ July, 2012].

[34] Object Management Group. MDA Specifications. `http://www.omg.org/mda/specs.htm`, 2001. [Online; accessed 12$^{\text{th}}$ April, 2012].

[35] John D. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *In In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.

[36] Stuart Kent. Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag.

[37] Object Management Group. Meta-Object Facility (MOF) 1.4 Specification. http://www.omg.org/spec/MOF/1.4/, April 2002. [Online; accessed April, 2012].

[38] AOSD Steering Committee. Aspect-Oriented Software Development Community & Conference. http://www.aosd.net/. [Online; accessed 24th July, 2012].

[39] Ronaldo Rodrigues Ferreira. Automatic code generation and solution estimate for object-oriented embedded software. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 909–910, New York, NY, USA, 2008. ACM.

[40] Dictionary and Thesaurus Merriam-Webster Online. http://www.merriam-webster.com. [Online; accessed July, 2012].

[41] Gonzalo Gnova. What is a metamodel: the OMG's metamodeling infrastructure. Doctorate Seminar, 2009.

[42] Emily Finn. The advantage of ambiguity. http://web.mit.edu/newsoffice/2012/ambiguity-in-language-0119.html, 2012. [Online; accessed 24th July, 2012].

[43] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[44] James Martin and Carma McClure. *Diagramming techniques for analysts and programmers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[45] D. Milicev. On the Semantics of Associations and Association Ends in UML. *Software Engineering, IEEE Transactions on*, 33(4):238 –251, april 2007.

[46] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[47] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[48] IEEE P1076 / VHDL Analysis and Standardization Group (VASG). VASG: VHDL Analysis and Standardization Group. http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome. [Online; accessed 25th July, 2012].

[49] Object Management Group. Object Management Group - UML. `http://www.uml.org/`. [Online; accessed 16th March, 2012].

[50] Nuno Amálio and Pierre Kelsen. VCL, a visual language for modelling software systems formally. In *Proceedings of the 6th international conference on Diagrammatic representation and inference*, Diagrams'10, pages 282–284, Berlin, Heidelberg, 2010. Springer-Verlag.

[51] Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Maria J. V. Pereira, Matej Črepinšek, Daniela da Cruz, and Pedro R. Henriques. Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. *Computer Science and Information Systems*, 7(2):247–264, May 2010.

[52] D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64 – 72, oct. 2004.

[53] John Howse, Steve Schuman, Gem Stapleton, and Ian Oliver. Diagrammatic Formal Specification of a Configuration Control Platform. *Electronic Notes in Theoretical Computer Science*, 259:87–104, December 2009.

[54] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[55] American Society of Mechanical Engineers. *ASME transactions*. Number 43. The Society, 1922.

[56] Object Management Group. *Object Constraint Language*. OMG, 2006.

[57] Ben Shneiderman and Pattie Maes. Direct manipulation vs. interface agents. *interactions*, 4(6):42–61, November 1997.

[58] Hiroshi Ishii. The tangible user interface and its evolution. *Communications of the ACM*, 51(6):32–36, June 2008.

[59] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, July 1999.

[60] Hiroshi Ishii and Brygg Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '97, pages 234–241, New York, NY, USA, 1997. ACM.

[61] Kenneth P. Fishkin. A taxonomy for and analysis of tangible interfaces. *Personal Ubiquitous Computing*, 8(5):347–358, September 2004.

[62] George Lakoff and Mark Johnson. *Metaphors We Live By*. University Of Chicago Press, 2nd edition, April 2003.

[63] Otmar Hilliges, Dominikus Baur, and Andreas Butz. A.: Photohelix: browsing, sorting and sharing digital photo collections. In *In: TABLETOP 2007: Second Annual IEEE International Workshop on Horizontal Interactive Human-Computer Systems*, pages 87–94, 2007.

[64] Crampton Smith. The hand that rocks the cradle. *Informatica Didactica*, 1995.

[65] John Underkoffler and Hiroshi Ishii. Urp: a luminous-tangible workbench for urban planning and design. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, CHI '99, pages 386–393, New York, NY, USA, 1999. ACM.

[66] Ben Piper, Carlo Ratti, and Hiroshi Ishii. Illuminating clay: a 3-d tangible interface for landscape analysis. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, CHI '02, pages 355–362, New York, NY, USA, 2002. ACM.

[67] Sergi Jordà, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. The reactable: exploring the synergy between live music performance and tabletop tangible interfaces. In *Proceedings of the 1st international conference on Tangible and embedded interaction*, TEI '07, pages 139–146, New York, NY, USA, 2007. ACM.

[68] Martin Kaltenbrunner and Ross Bencina. reactivision: a computer-vision framework for table-based tangible interaction. In *Proceedings of the 1st international conference on Tangible and embedded interaction*, TEI '07, pages 69–74, New York, NY, USA, 2007. ACM.

[69] Eva Hornecker and Jacob Buur. Getting a grip on tangible interaction: a framework on physical space and social interaction. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 437–446, New York, NY, USA, 2006. ACM.

[70] Eric Tobias, Eric Ras, and Nuno Amálio. VML Usability for Modelling TUI Scenarios - A Comparative Study. Technical Report TR-LASSY-12-06, University of Luxembourg, LASSY, 2012.

[71] John Hunt. Agile methods and the agile manifesto. In *Agile Software Construction*, pages 9–30. Springer London, 2006.

[72] Sharon Alayne Widmayer. Schema theory: An introduction. `http://tinyurl.com/98lx3yv`, 2005. [Online; accessed April, 2012].

[73] XJ Technologies Company. Simulation Software Tool - AnyLogic. `http://www.xjtek.com/`. [Online; accessed 5th March, 2012].

[74] Inc. Ventana Systems. Vensim. `http://www.vensim.com/`. [Online; accessed 5th March, 2012].

[75] Nicolas Genon, Daniel Amyot, and P. Heymans. Analysing the Cognitive Effectiveness of the UCM Visual Notation. *System Analysis and Modeling: About Models*, 6598/2011:221–240, 2011.

[76] D Varro. Towards Symbolic Analysis of Visual Modeling Languages. *Electronic Notes in Theoretical Computer Science*, 72(3):51–64, February 2003.

[77] Andrew Fish, Jean Flower, and John Howse. The semantics of augmented constraint diagrams. *Journal of Visual Languages & Computing*, 16(6):541–573, December 2005.

[78] Joseph Yossi Gil, John Howse, and Stuart Kent. Constraint Diagrams: A Step Beyond UML. In *Technology of Object-Oriented Languages and Systems (TOOLS USA'99)*, Santa Barbara, California , USA, 1999.

[79] Robert Muetzelfeldt and Jon Massheder. The Simile visual modelling environment. *European Journal of Agronomy*, 18(3-4):345–358, January 2003.

[80] S. Sadi and P. Maes. subTextile: Reduced event-oriented programming system for sensate actuated materials. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, pages 171–174, September 2007.

[81] James Lin, Michael Thomsen, and J.A. Landay. A visual language for sketching large and complex interactive designs. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, number 4, pages 307–314, New York, New York, USA, 2002. ACM.

[82] A. Schurr, A Winter, and A. Zundorf. Visual programming with graph rewriting systems. In *Visual Languages, Proceedings., 11th IEEE International Symposium on*, pages 326–333, Darmstadt , Germany, 1995. IEEE.

[83] L. Spratt and A. Ambler. A visual logic programming language based on sets and partitioning constraints. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 204–208. IEEE, 1993.

[84] Margaret Burnett, John Atwood, R. Walpole Djang, J. Reichwein, H. Gottfried, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2):155–206, 2001.

[85] D. Lucanin and Ivan Fabek. A visual programming language for drawing and executing flowcharts. In *MIPRO, 2011 Proceedings of the 34th International Convention*, pages 1679–1684. IEEE, 2011.

[86] Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307, June 2004.

[87] KD Swenson. A visual language to describe collaborative work. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 298–303, Bergen, 1993. IEEE.

[88] Roswitha Bardohl, Hartmut Ehrig, J. De Lara, and G. Taentzer. Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. *Fundamental Approaches to Software Engineering*, pages 214–228, 2004.

[89] I. Burnett, M.J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. Scaling Up Visual Programming Languages. *Computer*, 28(3):45–54, March 1995.

[90] Akos Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, 2863:92–95, 2003.

[91] Sara Brockmans, Raphael Volz, and Andreas Eberhart. Visual modeling of OWL DL ontologies using UML. In *The Semantic WebISWC 2004*, pages 198–213. Springer, 2004.

[92] Terry Halpin. Object-role modeling: an overview. `http://www.orm.net/pdf/ORMwhitePaper.pdf`, 1998. [Online; accessed March, 2012].

[93] Emmanuel Chailloux and Philippe Codognet. Toward visual constraint programming. *Visual Languages, 1997. Proceedings. 1997 IEEE Symposium on*, pages 420–421, 1997.

[94] C. Kiesner, G. Taentzer, and J. Winkelmann. Visual OCL: A Visual Notation of the Object Constraint Language. Technical Report 2002/23, Technical University of Berlin, 2002.

[95] Institut für Softwaretechnik und Theoretische Informatik. Visual OCL. `http://tfs.cs.tu-berlin.de/vocl/`. [Online; accessed 6$^{\text{th}}$ March, 2012].

[96] Paolo Bottoni, Manuel Koch, Francesco Parisi-presicce, and Gabriele Taentzer. A Visualization of OCL using Collaborations. *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, 2185:257–271, September 2001.

[97] N. Amalio and Pierre Kelsen. Modular design by contract visually and formally using VCL. In *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*, pages 227–234. IEEE, September 2010.

[98] Nuno Amáio, Pierre Kelsen, Qin Ma, and Christian Glodt. Using VCL as an aspect-oriented approach to requirements modelling. *Transactions on Aspect-Oriented Software Development*, 7:151–199, 2010.

[99] Nuno Amálio, Christian Glodt, and Pierre Kelsen. Building VCL Models and Automatically Generating Z Specifications from Them. *FM 2011: Formal Methods*, 6664:149–153, 2011.

[100] Stuart Kent. Constraint diagrams: visualizing invariants in object-oriented models. In *OOPSLA '97 Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 32. ACM, 1997.

[101] G. Stapleton. A Decidable Constraint Diagram Reasoning System. *Journal of Logic and Computation*, 15(6):975–1008, December 2005.

[102] Executable Visual Contracts. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 63–70. IEEE, 2005.

[103] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992.

[104] Paul Bicheler. A toolkit for table-based tangible widgets. Master's thesis, University of Luxembourg, 2012.

[105] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[106] Nuno Amálio and Pierre Kelsen. The visual contract language: abstract modelling of software systems visually, formally and modularly. Technical Report TR-LASSY-10-03, University of Luxembourg, 2010.

[107] Nuno Amálio, Fiona Polack, and Susan Stepney. An object-oriented structuring for Z based on views. In *Proceedings of the 4th international conference on Formal Specification and Development in Z and B*, ZB'05, pages 262–278, Berlin, Heidelberg, 2005. Springer-Verlag.

[108] Nuno Amálio. *Generative frameworks for rigorous model-driven development*. PhD thesis, University of Kent, 2007.

[109] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International Ltd., Hertfordshire, UK, 1992.

[110] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[111] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.

[112] Anneke Kleppe. Software Language Engineering. In *The Field of Software Language Engineering*, pages 1–7. Springer-Verlag, Berlin, Heidelberg, 2009.

[113] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[114] Vinay Kulkarni and Sreedhar Reddy. Separation of concerns in model-driven development. *IEEE Software*, 20(5):64–69, September 2003.

[115] Sifteo Inc. Sifteo - Intelligent Play. `https://www.sifteo.com/`. [Online; accessed 2$^{\text{nd}}$ August, 2012].

[116] David Merrill, Jeevan Kalanithi, and Pattie Maes. Siftables: towards sensor network user interfaces. In *Proceedings of the 1st international conference on Tangible and embedded interaction*, TEI '07, pages 75–78, New York, NY, USA, 2007. ACM.

[117] Laurent Kirsch, Jean Botev, and Steffen Rothkugel. An extensible tool set for creating and connecting reusable learning resources. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2012*, pages 1434–1442. AACE, June 2012.

[118] Laurent Kirsch, Jean Botev, and Steffen Rothkugel. The snippet system - reusing and connecting documents. In *Proceedings of the 7th International Conference on Digital Information Management*, ICDIM 2012. IEEE, 2012. To appear.

[119] Eric Tobias, Eric Ras, and Nuno Amálio. Suitability of Visual Modelling Languages for Modelling Tangible User Interface Applications. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 2012. To appear.

[120] Nuno Amálio, Qin Ma, Christian Glodt, and Pierre Kelsen. VCL specification of the car-crash crisis management system. Technical Report TR-LASSY-09-03, University of Luxembourg, LASSY, 2009.

[121] Jérôme Leemans and Nuno Amálio. A VCL Model of a Cardiac Pacemaker. Technical Report TR-LASSY-12-04, University of Luxembourg, LASSY, 2012.

[122] Jérôme Leemans and Nuno Amálio. Modelling a Cardiac Pacemaker Visually and Formally. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 2012. To appear.

[123] Nuno Amálio. The VCL Model of bCMS. Technical Report TR-LASSY-12-09, University of Luxembourg, 2012.

[124] Stephen Mellor. Models. Models. Models. So What? In *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 1–1. Springer Berlin / Heidelberg, 2009.

Appendix A

TUI metamodel VCL packages

The following sections present all Visual Contract Language (VCL) packages needed to build the TUI metamodel based on the work of Paul Bicheler in the scope of his Mater's Thesis [104]. The specification document considered for the modelling is an early version of his thesis. As explained in Section 5.2.2, due to concurrency in the elaboration of the theses, function safety, selector endpoints as well as system interactions were not modelled. The packages will detail all diagrams. Each section will provide an overview of the package and complement the general description given in Section 5.4.

For an introduction on VCL, please consider reading Section 4. In this appendix, the following notation is used; Structural Diagrams (SD), Behavioural Diagrams (BD), Package Diagrams (PD), Assertion Diagrams (AD), Contract Diagrams (CD). Table A.1 lists all VCL packages and the corresponding figures for each VCL package.

Table A.1: TUI metamodel VCL packages and figures

Package name	Figures
Pimitives A.1	PD A.1 SD A.2
Actions A.2	PD A.3 SD A.4 BD A.5 EndPointRequired AD A.6 Action_Default CD A.7 CreateDefaultAction CD A.8 AddMapping CD A.9

Continued on next page

Table A.1 – *Continued from previous page*

Package name	Figures
Attributes A.3	PD A.10 SD A.11 BD A.12 IsUnique AD A.13 Attribute_NewID CD A.14 Attribute_New CD A.15 Attribute_Update CD A.16 Attribute_Delete AD A.17 CreateIDAttribute CD A.18 CreateDefaultAttribute CD A.19
Effects A.4	PD A.20 SD A.21 BD A.22 EndPointRequired AD A.23 Effect_Default AD A.24 CreateDefaultEffect CD A.47 AddMapping CD A.26
EndPoints A.5	PD A.27 SD A.28 BD A.29 HandleIsRoot AD A.30 EndPoint_Default CD A.31 CreateDefaultEndPoint CD A.32 Zone_Default CD A.33 Zone_Delete AD A.34 WidgetComponent_Default CD A.35
Functions A.6	PD A.36 SD A.37 BD A.38 Function_Default CD A.39 CreateDefaultFunction CD A.40 AddActionToEffect CD A.41 AddActionToEndPoint CD A.42 AddAction CD A.43
Mappings A.7	PD A.44 SD A.45 BD A.46 Mapping_New CD A.49 CreateNewMapping CD A.48

Continued on next page

Table A.1 – *Continued from previous page*

Package name	Figures
Layers A.8	PD A.50 SD A.51 BD A.52 AddZone CD A.53 RemoveZone CD A.54 AddWidget CD A.55
Widgets A.9	PD A.56 SD A.57 BD A.58 Widget_Default CD A.59 Identity_Default CD A.60 AddAction CD A.61
TUI A.10	PD A.62

A.1 The TUIPrimitives package

This package specifies a boolean primitive, `Bool` which can take the values `True` and `False`. It also specifies `Strings` and `Accessors`, a subset of `Strings` as shown by the insideness on the SD. `Strings` in turn are a subset of `Element`. As by the specification, `Attributes` can take anything as value. As such, `Element` encompasses all other types and is not defined to be limited to those subsets of types. In the final application, `Element` should be merged with the root type of all system primitives.

Figure A.1: The *`TUIPrimitives'`* package diagram

A.2 The Actions package

An `Action` evaluates a `Condition`. Please note that the `Condition` was not modelled beyond the concept as it was unclear for a long time what form the `Conditions` would take and if there were concrete pre- and postconditions or not. An `Action` may have `Attributes`, one of which must be an `id`. `Actions` can trigger `Effects` or immediately produce a response in an `EndPoint`. `Actions` also include a reference to one or several `Mappings`, mapping

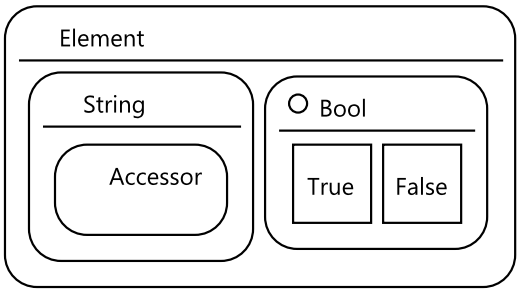


Figure A.2: The *TUIPrimitives'* structure diagram

their `Attributes` to either `Effects` or `EndPoints`. The `EndPointRequired` invariant partially models the requirement that a `Function` needs to have an `EndPoint`. While this is theoretically covered by the cardinalities in the *Function* package, the local cardinalities would not oblige the `Action` to be mapped to either an `Effect` or an `EndPoint`. The package contains two global operations, one for creating a default `Action` and one for adding a `Mapping`. To create a default `Action`, the global operation calls upon the `Default` constructor of the `Action`.

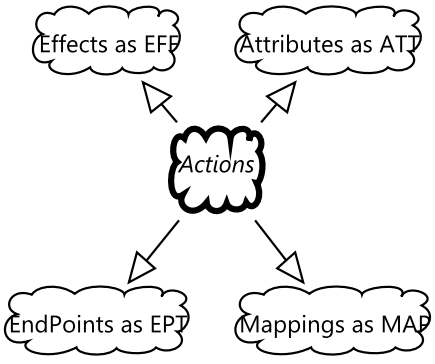


Figure A.3: The `Actions'` package diagram

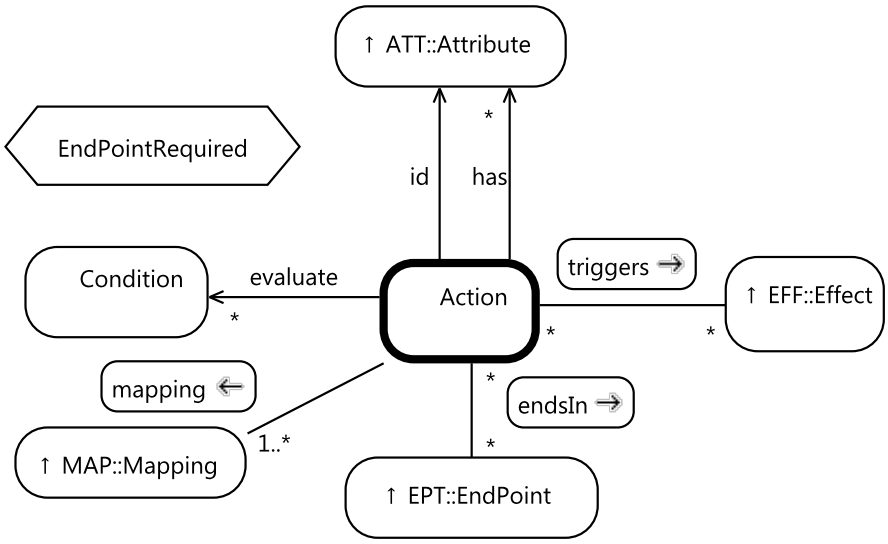


Figure A.4: The `Actions'` structure diagram

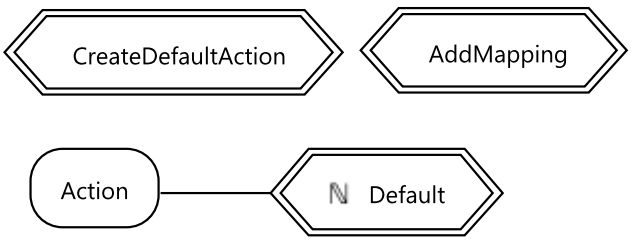


Figure A.5: The `Actions'` behaviour diagram

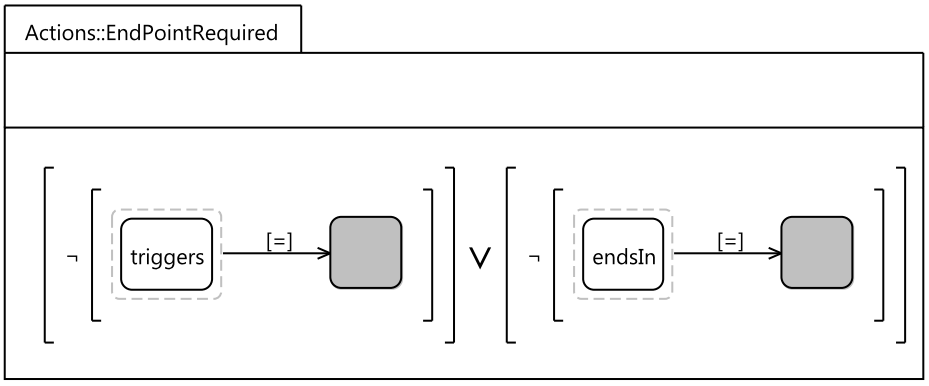


Figure A.6: The `EndPointRequired` assertion expressing a local invariant on the `Action's` makeup

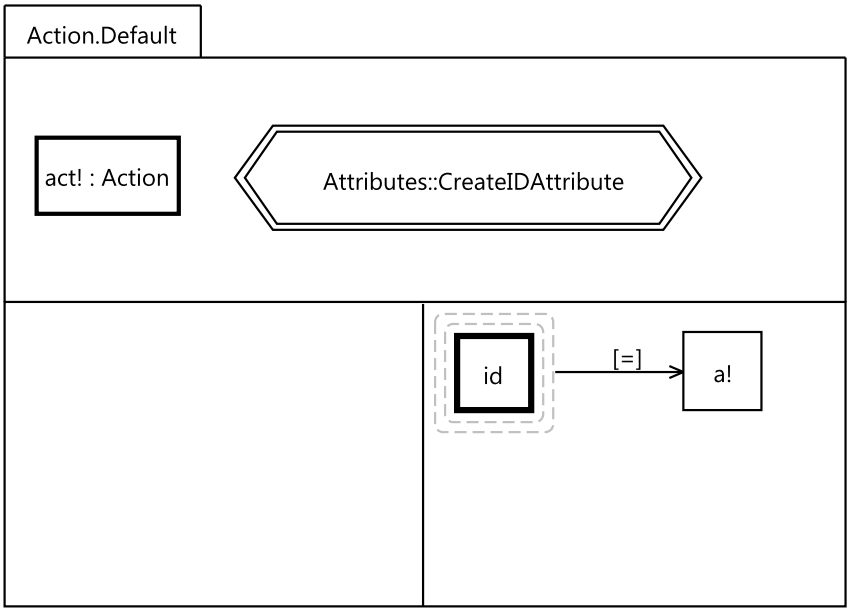


Figure A.7: The `Action's Default` operation

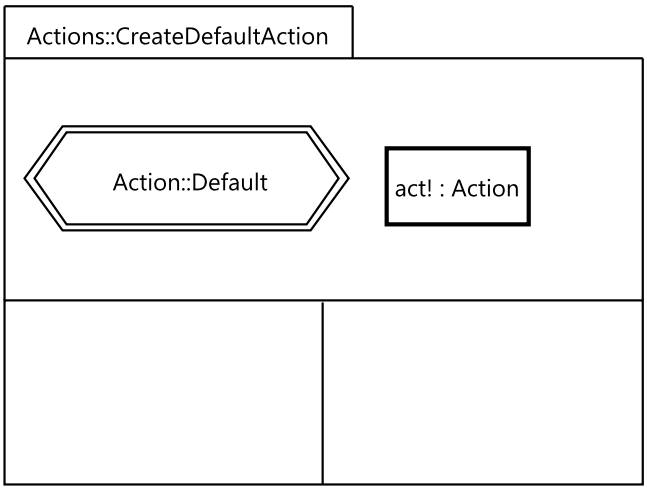


Figure A.8: The *Mapping's* `CreateDefaultAction` operation

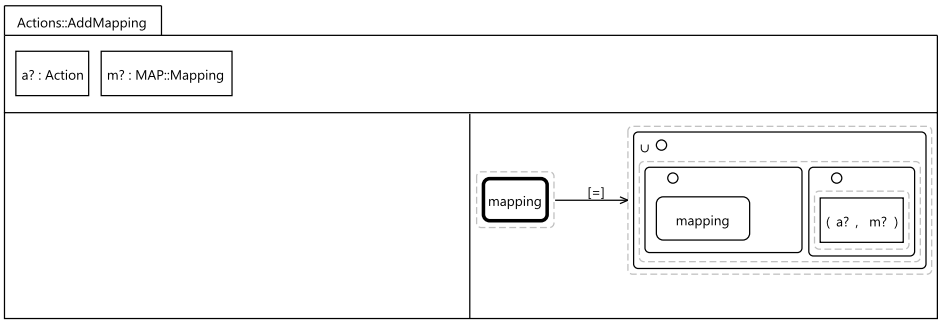


Figure A.9: The *Action's* `AddMapping` operation

A.3 The Attributes package

The *Attributes* VCL package models the `Attribute` as a `key-_value` pair, binding an `Accessor` to an `Element`. The diagram also defines a constant, a special `Accessor` that is used for identifiers. The BD diagram holds the behaviour of `Attributes` as described and implied by the specification. The requirement that all `Attributes` must have a unique ID is enforced through the `CreateIDAttribute` operation which all model elements do use by default to get an identifier. The operation uses the `IsUnique` assertion to verify that the new identifier is not yet present in the system. The `CreateDefault-Attribute` global operation is used to construct a default `Attribute` to be used in the mapping of a default `Function`.

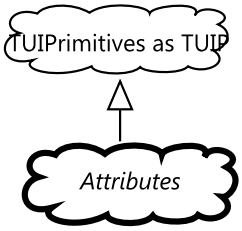


Figure A.10: The *Attributes'* package diagram

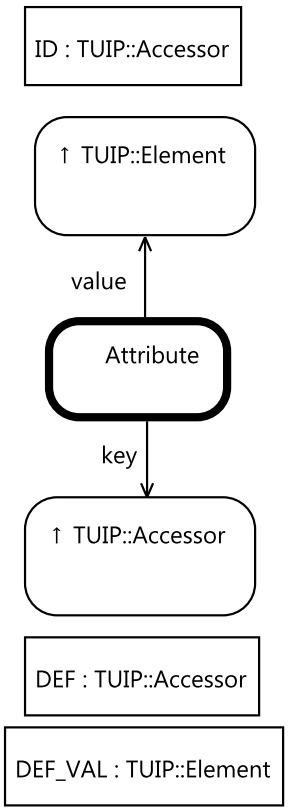


Figure A.11: The *Attributes'* structure diagram

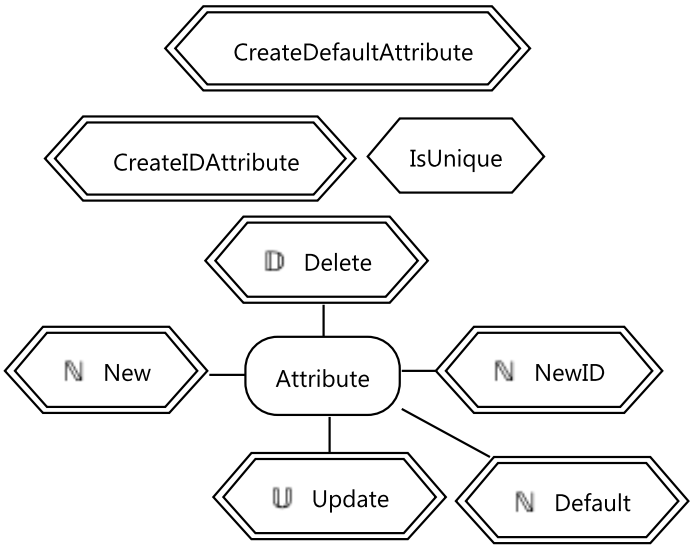


Figure A.12: The *Attributes'* behaviour diagram

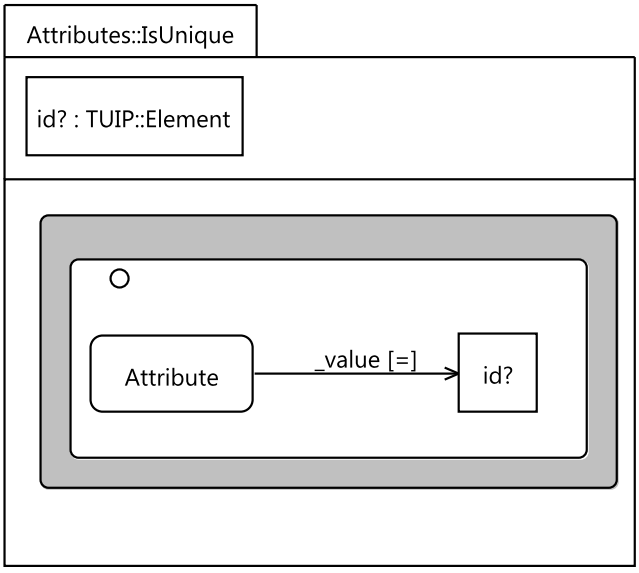


Figure A.13: The `IsUnique` assertion expressing constraints on the uniqueness of `UUIDs`

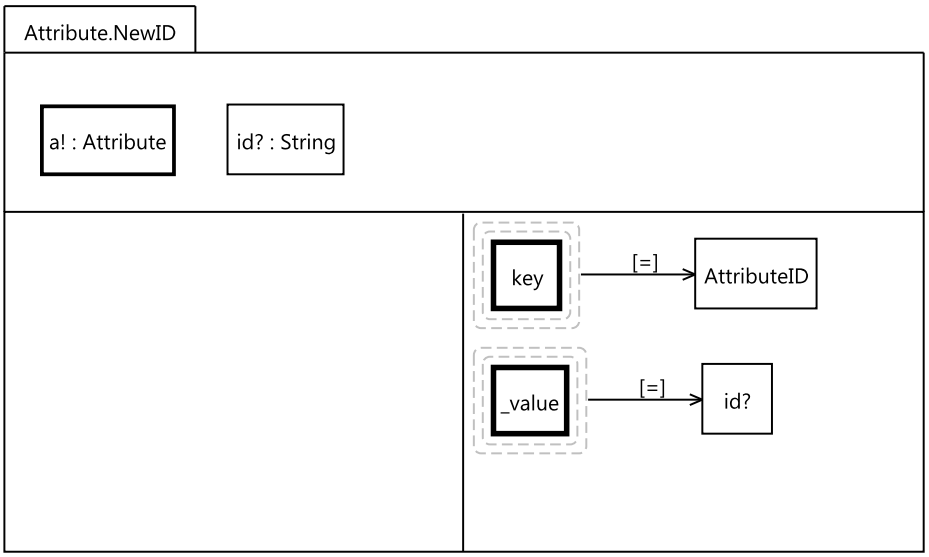


Figure A.14: The `Attribute's NewID` operation

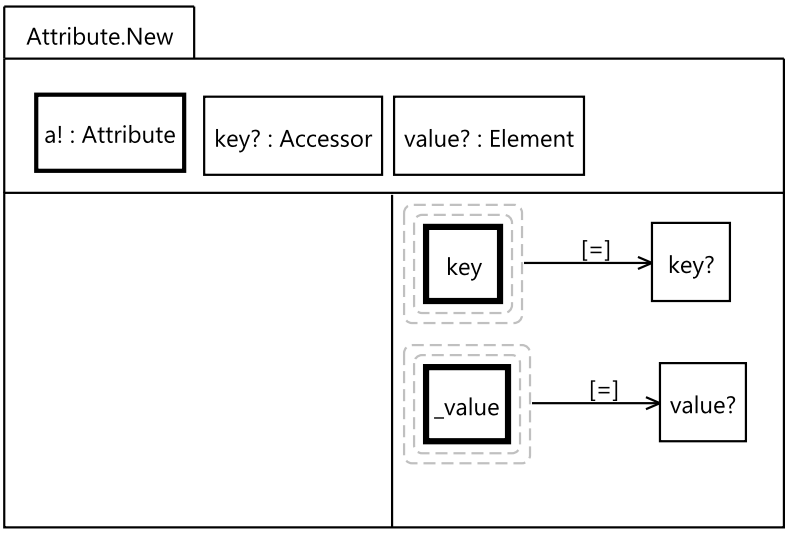


Figure A.15: The `Attribute's New` operation

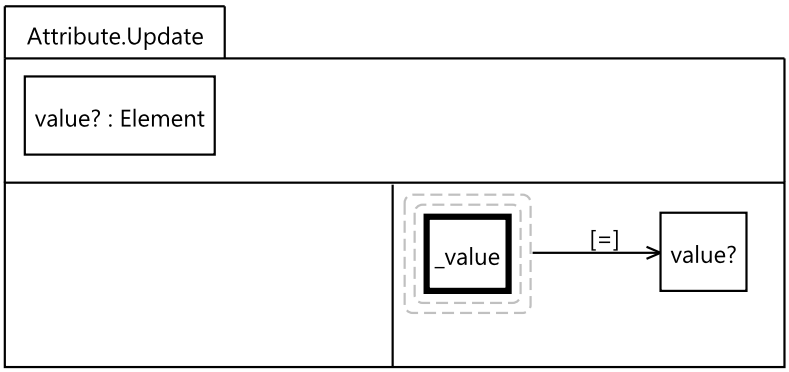


Figure A.16: The `Attribute's Update` operation

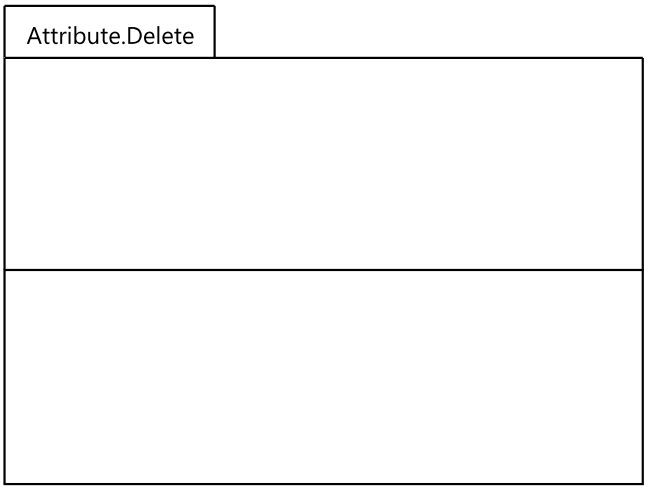


Figure A.17: The `Attribute's Delete` operation

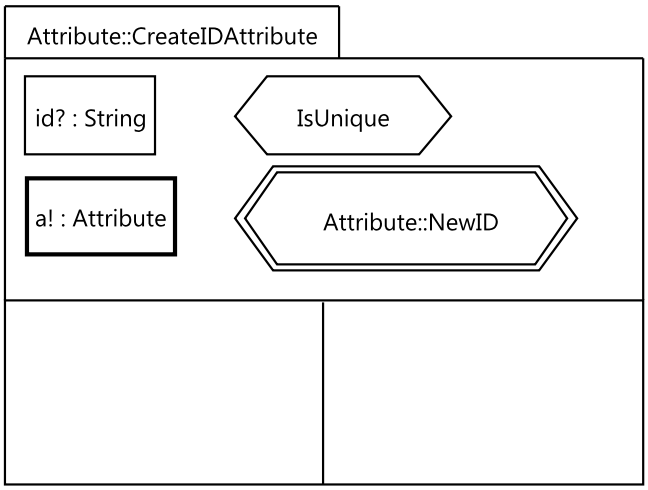


Figure A.18: The `Attribute's CreateIDAttribute` operation

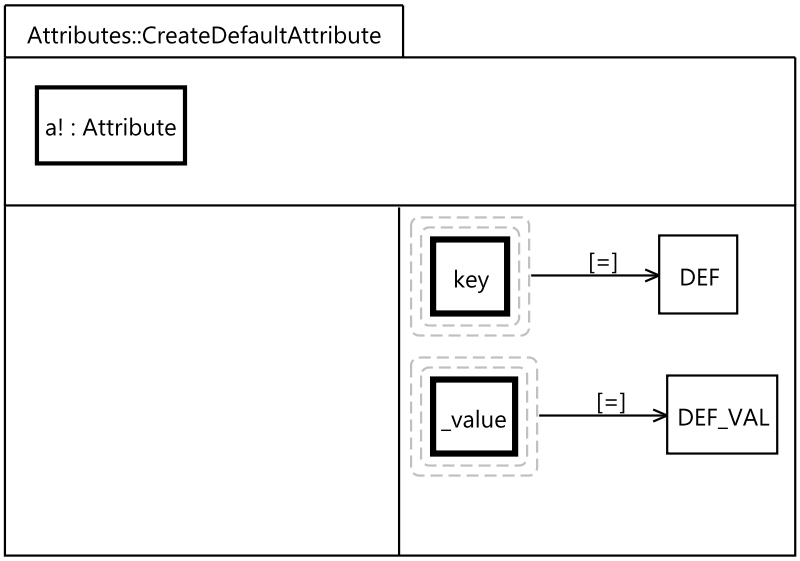


Figure A.19: The *`Attribute's`* `CreateDefaultAttribute` operation

A.4 The Effects package

`Effects` may have several `Attributes` in addition to one `id Attribute`. `Effects` evaluate conditions before executing and propagate their result by a `Mapping` of their `Attributes` to either another `Effect` or to an `EndPoint`. The `EndPointRequired` invariant, a variant of the `Action's` identi-

cally named method, enforces that an `Effect` propagates to either an `Effect` or an `EndPoint` but not both. This could not be expressed by the cardinalities alone. The package contains two global operations, one for creating a default `Effect` and one for adding a `Mapping`. To create a default `Effect`, the global operation calls upon the `Default` constructor of the `Effect`.

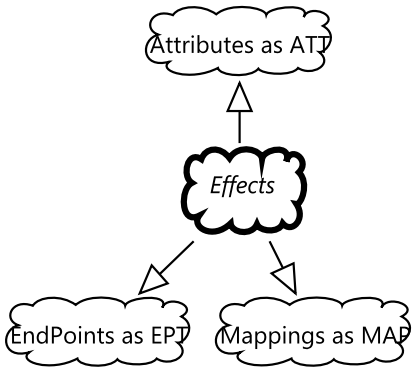


Figure A.20: The `Effects'` package diagram

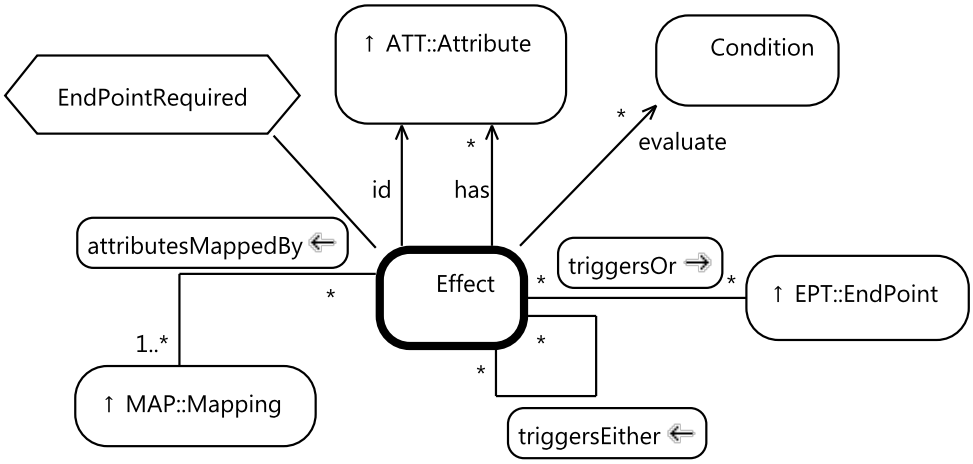


Figure A.21: The `Effects'` structure diagram

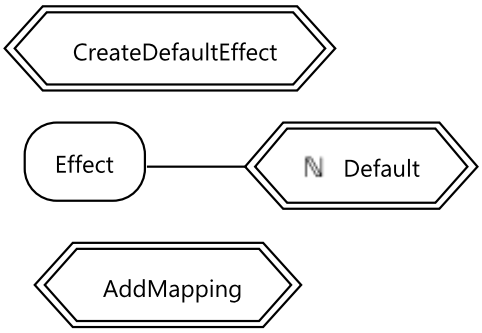


Figure A.22: The `Effects'` behaviour diagram

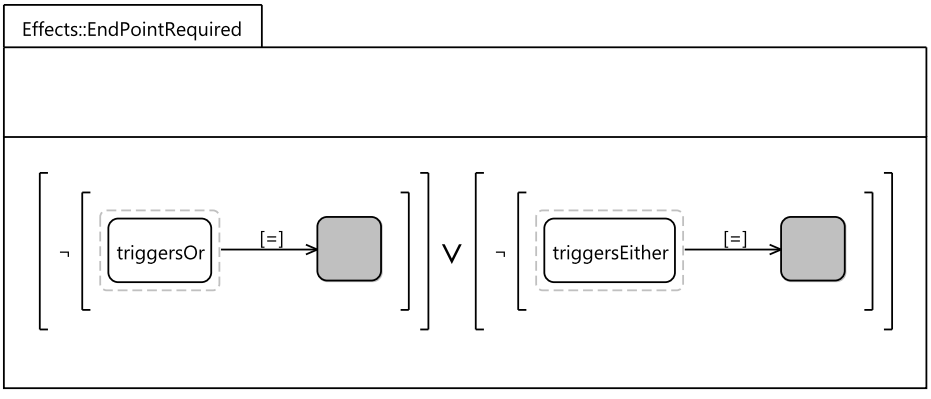


Figure A.23: The `EndPointRequired` assertion expressing a local invariant on the `Effect's` makeup

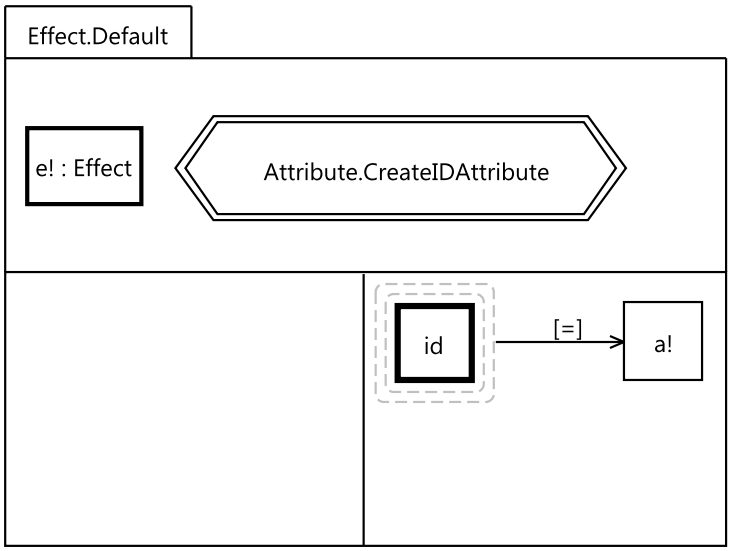


Figure A.24: The `Effect's Default` operation

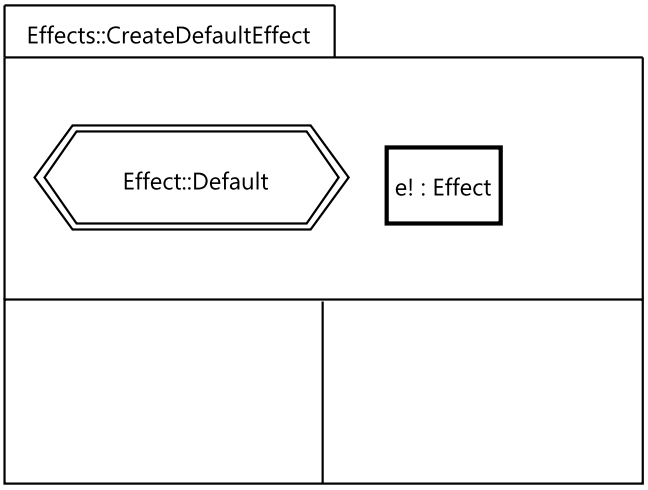


Figure A.25: The *`Effect's`* `CreateDefaultEffect` operation

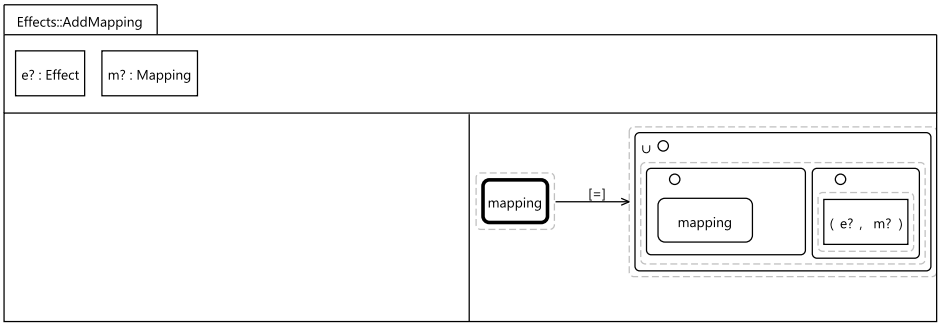


Figure A.26: The *Effect's* `AddMapping` operation

A.5 The EndPoints package

`EndPoints` are part of the widget structure. They are containers for `Zones` and `WidgetComponents` to enable binding and selection. The different components are described in 5.2.2. The requirement that all `Endpoints` must have at least `Attribute` is fulfilled by using a cardinality on a parameter edge. As identifiers are generated by default, they have to be present unless removed. This would ideally be covered by an invariant but unfortunately, quantifiers which would enable the expressions of that constraint are not yet implemented in VCL. The constraint that a `Handle` cannot be composed of other `Handles` is checked by the `handleIsRoot` invariant.

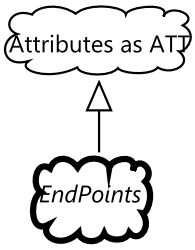


Figure A.27: The `EndPoints'` package diagram

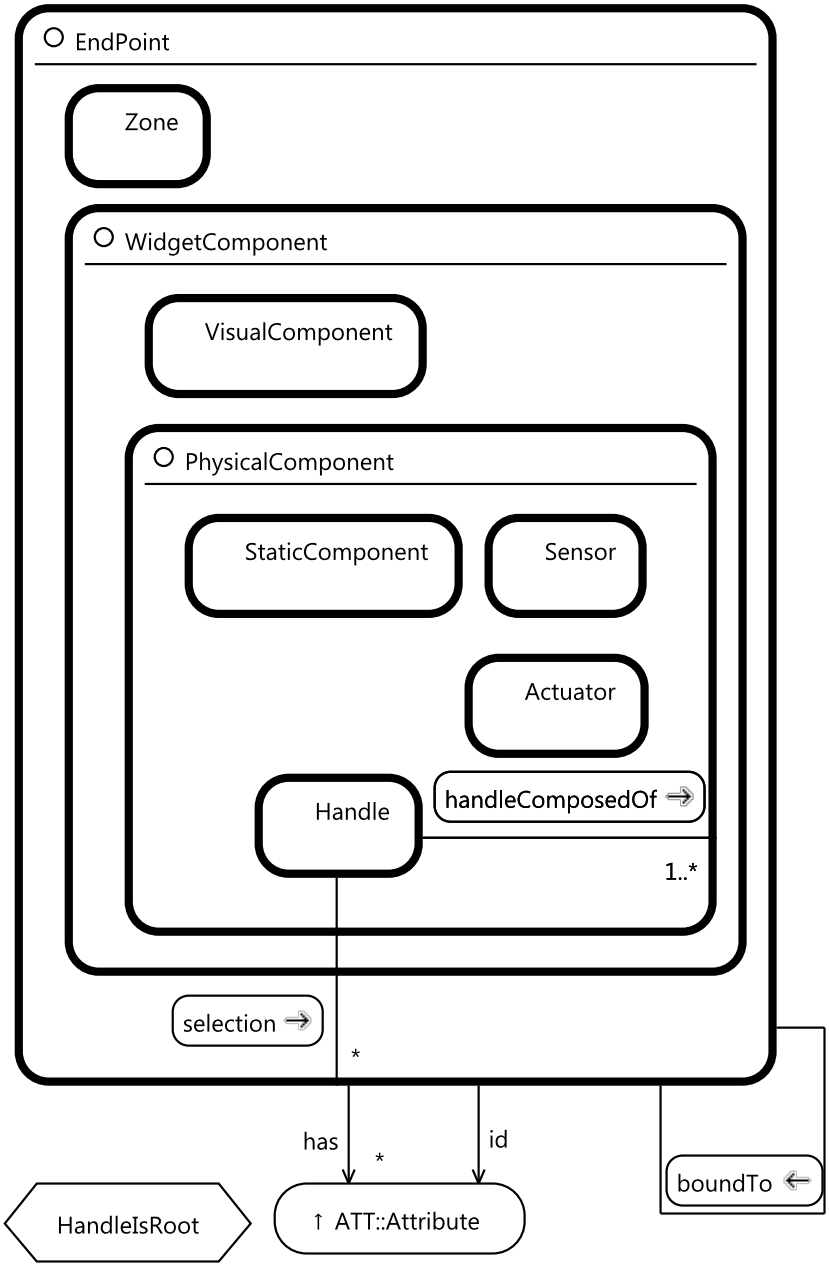


Figure A.28: The `EndPoints'` structure diagram

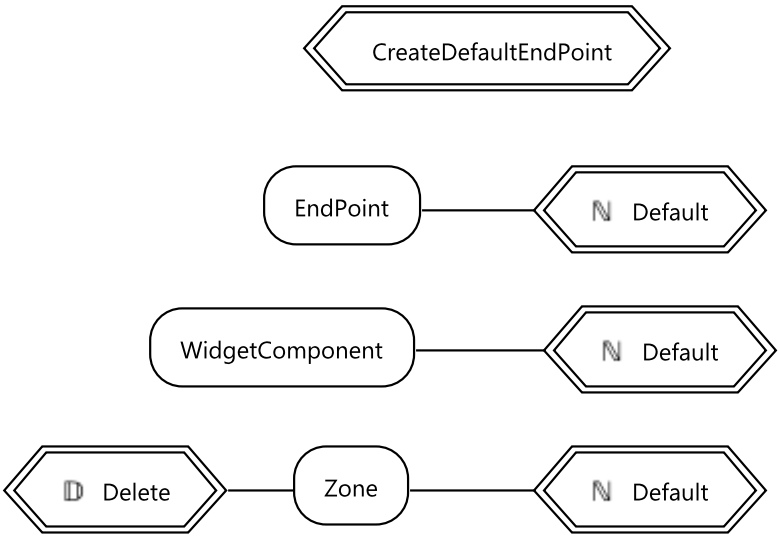


Figure A.29: The `EndPoints'` behaviour diagram

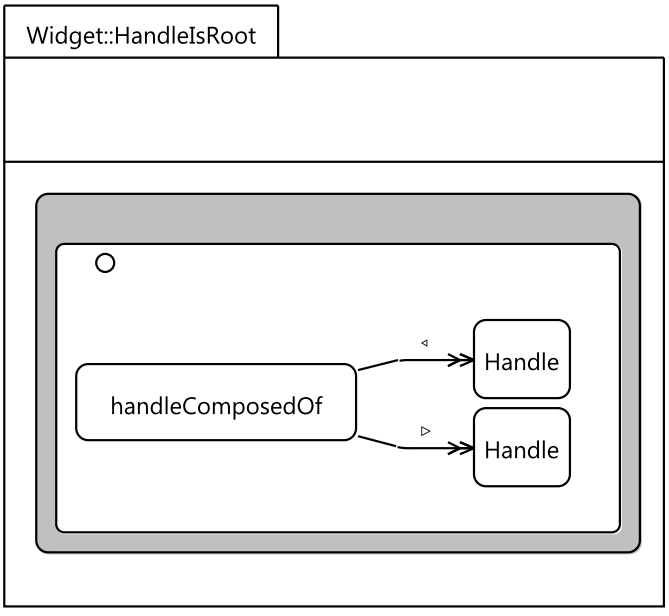


Figure A.30: The `HandleIsRoot` assertion expressing constraints on the composition of `WidgetComponents`

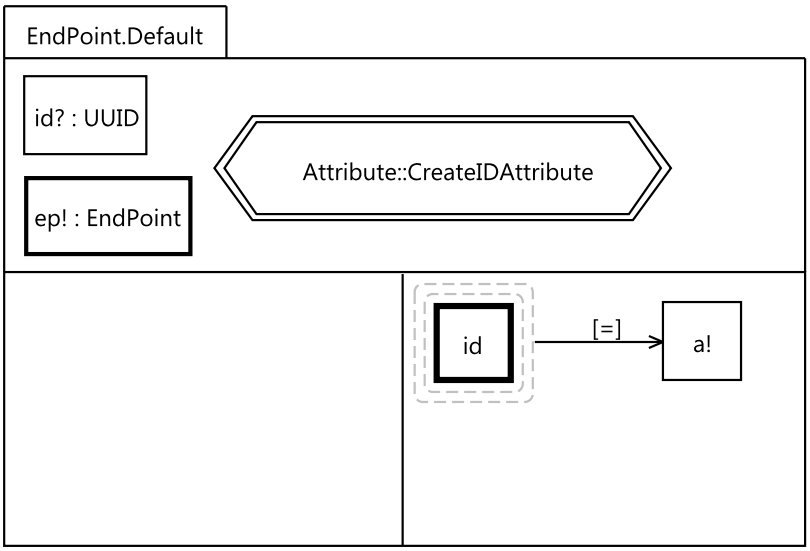


Figure A.31: The `EndPoint's Default` operation

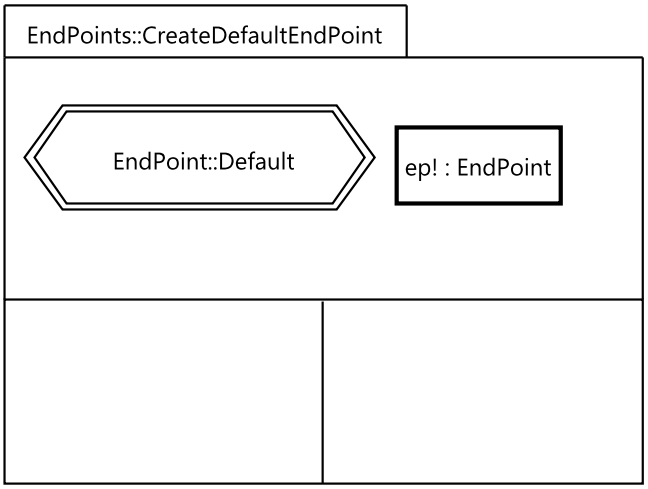


Figure A.32: The `Mapping's CreateDefaultEndPoint` operation

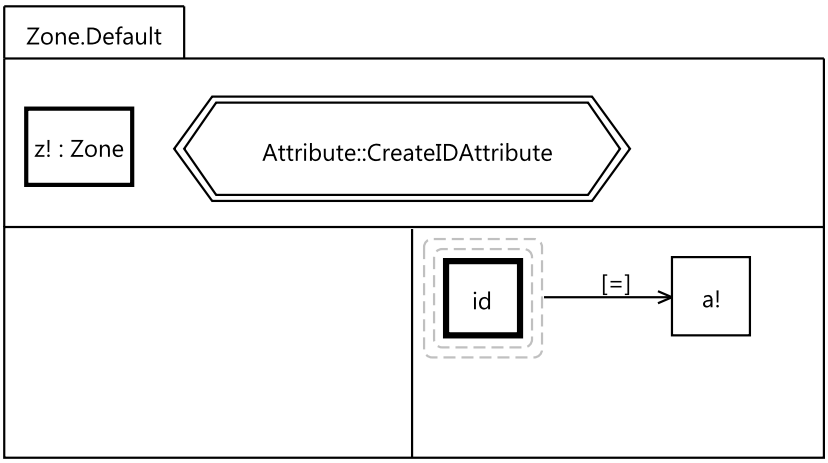


Figure A.33: The `Zone's Default` operation

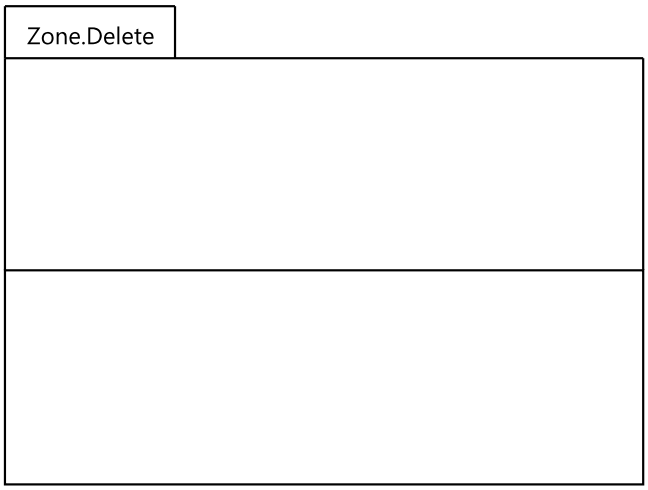


Figure A.34: The `Zone's Delete` operation

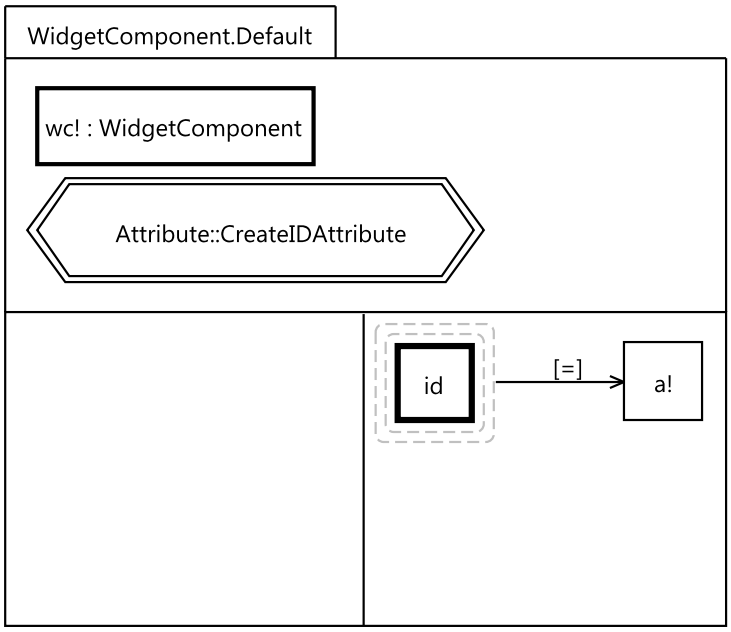


Figure A.35: The `WidgetComponent's Default` operation

A.6 The Functions package

A `Function` expresses the widget's need for a triggering `Action`, any number of optional `Effects` and a mandatory `EndPoint`. In addition to specifying the `Function's` structure, several contracts are defined in the ***`Function`*** package. The `CreateDefaultFunction` operation satisfies the requirement that there should be possible to have a default `Function`. Moreover, the package defines the concept of `FunctionElement`, a superset for `Actions`, `Effects` and `EndPoints`. A few token operations have been modelled.

The `AddAction` global operation is called upon by a `Widget's` identically named global operation. This is due to TWT specifying that the user works at `Widget` level. `AddAction` uses non-determinism to execute either `AddActionToEffect` or `AddActionToEndPoint`. These operations add all required tuples, as well as constructing the `Mapping` of `Attributes`. Ideally, additional invariants should be introduced, guaranteeing that all `Actions` eventually end in an `EndPoint`. However, with the possible looping of `Effects` it is currently not possible to formulate such an invariant. An operator to formulate transitive closures would be one option to realise such an invariant. More on this matter can be read in the conclusion in Section 5.10.

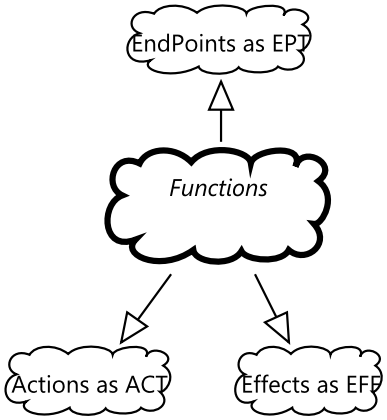


Figure A.36: The *Functions'* package diagram

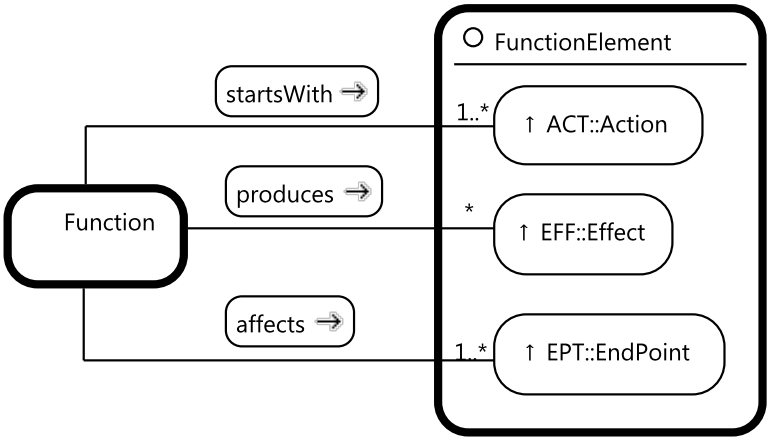


Figure A.37: The *Functions'* structure diagram

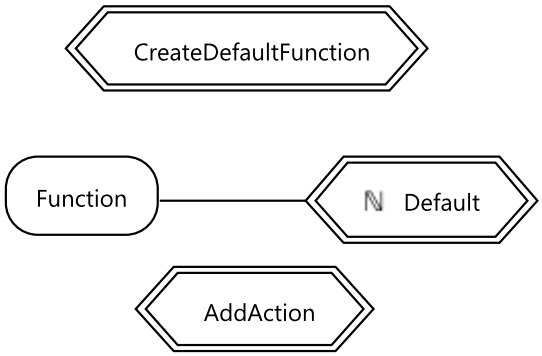


Figure A.38: The *Functions'* behaviour diagram

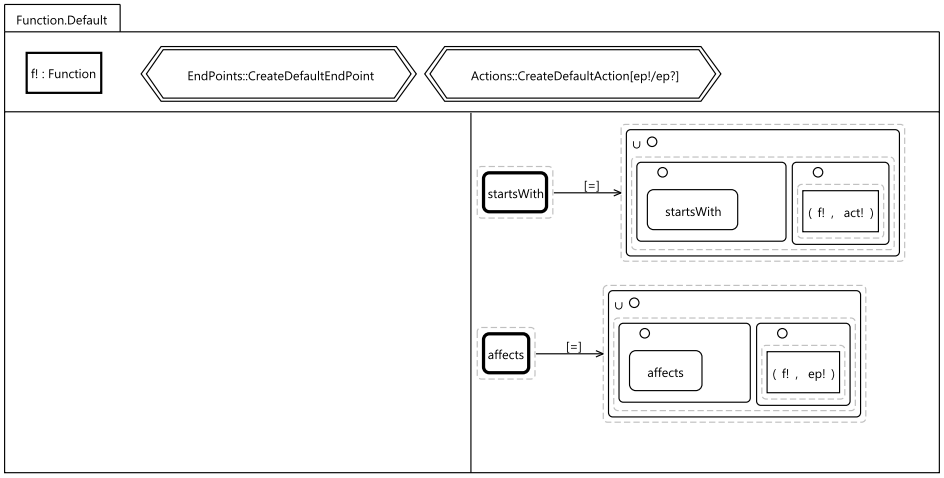


Figure A.39: The `Function's Default` operation

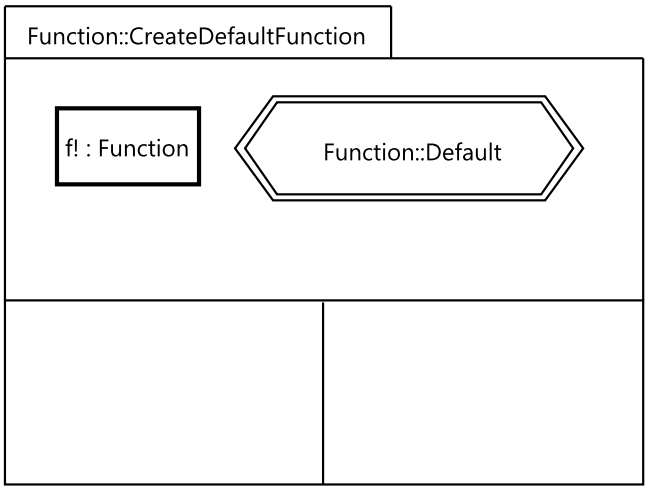


Figure A.40: The `Function's CreateDefault` operation

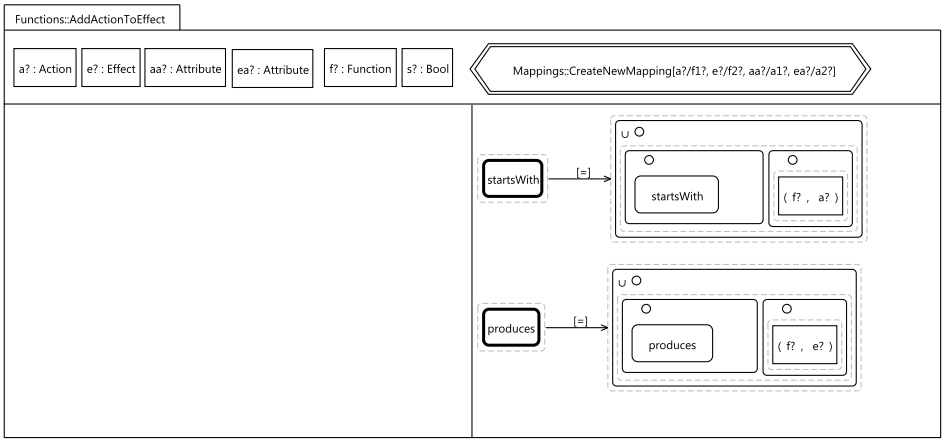


Figure A.41: The `Function's AddActionToEffect` operation

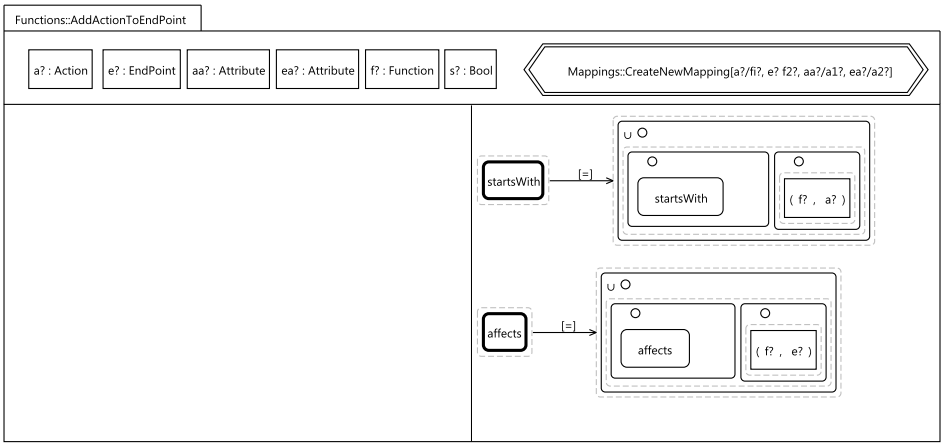


Figure A.42: The `Function's AddActionToEndPoint` operation

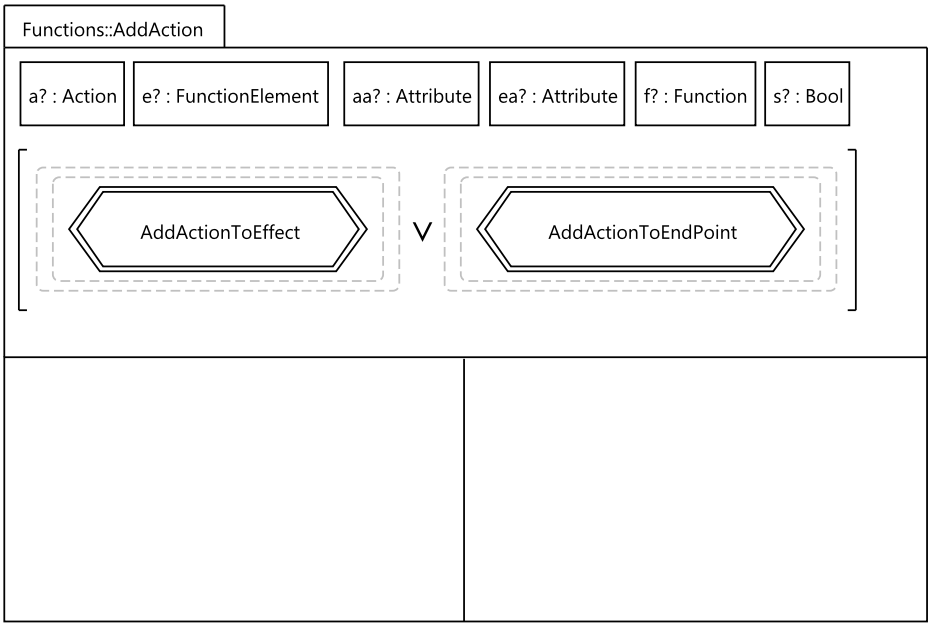


Figure A.43: The `Function's AddAction` operation

A.7 The Mappings package

The *`Mapping`* package defines how `FunctionElements'` input and output `Attributes` are laced together. The implicit constraint of only mapping

`Attributes` from and to `FunctionElements` having them is satisfied on the `CreateNewMapping` operation . The requirement that both parametrised `Attributes` must stem from different `FunctionElements` is modelled as two preconditions. While this is also specified at ***`Widget`*** level, it is best to express conditions on every concern they apply. Note that the specification did not specify a separation of `Attributes` into input and output which would have given room to a more correct model.

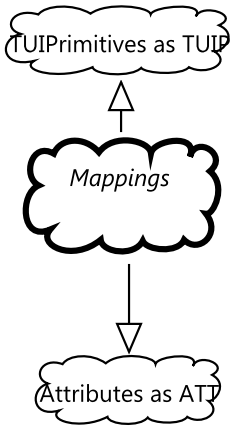


Figure A.44: The ***`Mappings'`*** package diagram

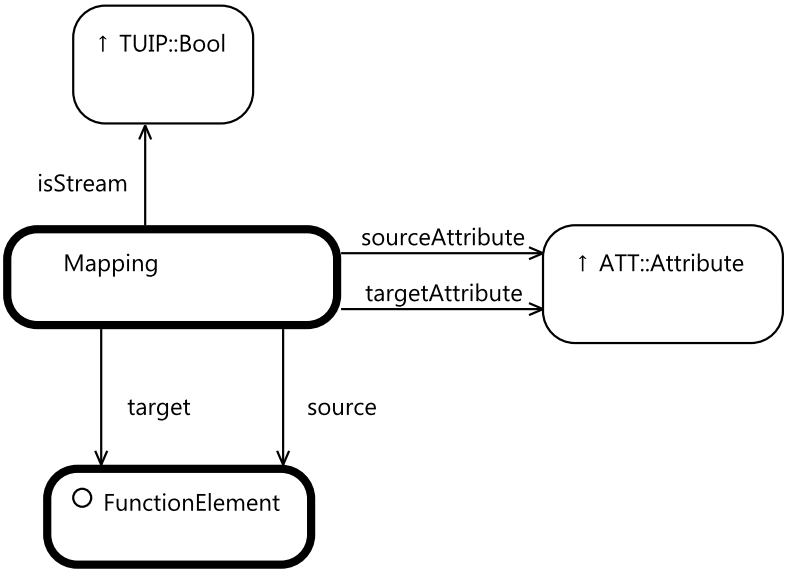


Figure A.45: The ***`Mappings'`*** structure diagram

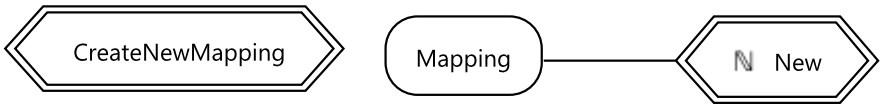


Figure A.46: The *Mappings'* behaviour diagram

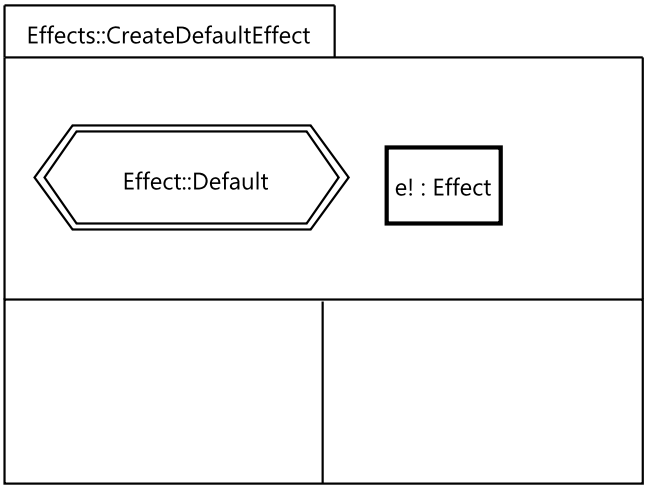


Figure A.47: The `Mapping's CreateDefaultEffect` operation

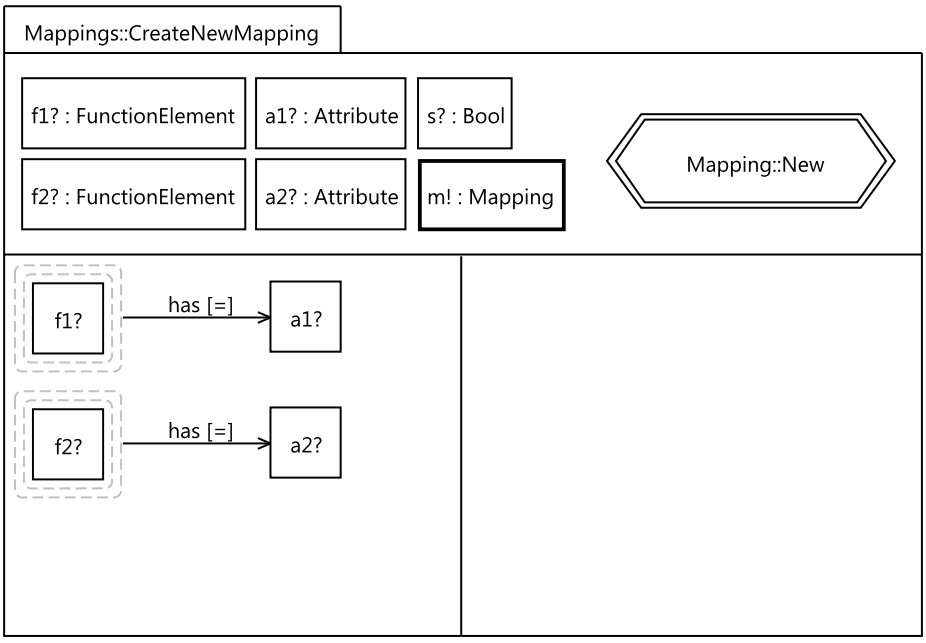


Figure A.48: The `Mapping's CreateNewMapping` operation

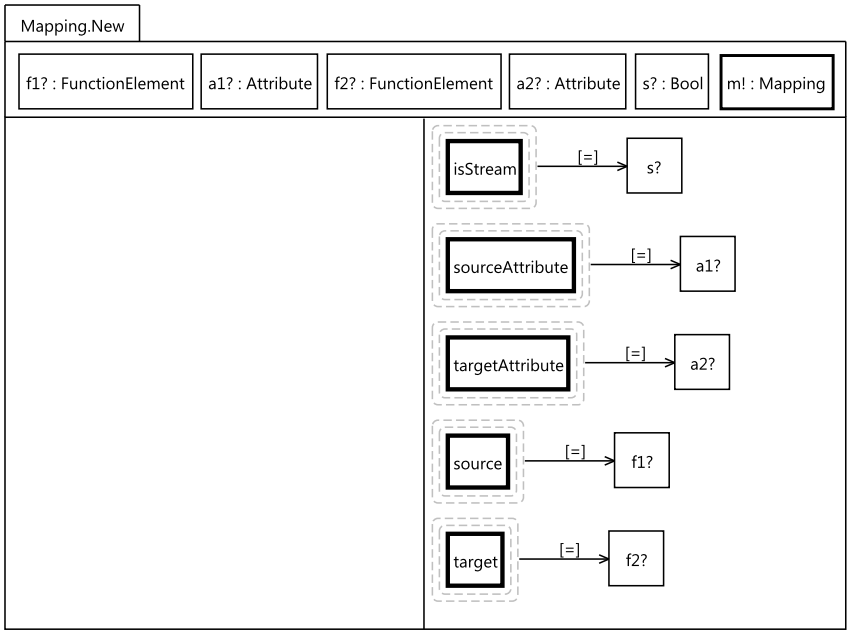


Figure A.49: The `Mapping's New` operation

A.8 The Layers package

The *Layers* package defines two `Layers`, the `TangibleLayer` holding all `Widgets` and the `ApplicationLayer` exposing all `Zones` defined by the application. The behaviour is implicitly read from the specification. As the goal of TWT is to create tangible widgets, there must be a way to add `Widgets`. Furthermore, for them to interact with an application, there must be a way for it to add `Zones`. The consequences of a `Widget's` removal from the `TangibleLayer` depend on the sensor capabilities of the physical table and the behaviour remains, therefore, unspecified at a metamodel level. The specification document states that the lifecycle of a `Zone` ends when it is removed from the `ApplicationLayer`. This is enforced by the `RemoveZone` operation which will ensure the deletion of the `Zone`.

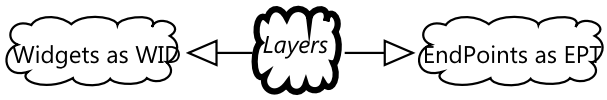


Figure A.50: The *Layers'* package diagram

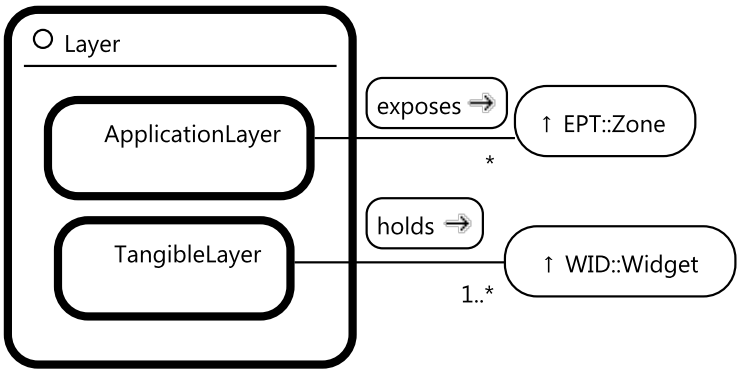


Figure A.51: The *Layers'* structure diagram

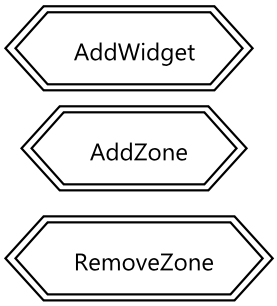


Figure A.52: The *Layers'* behaviour diagram

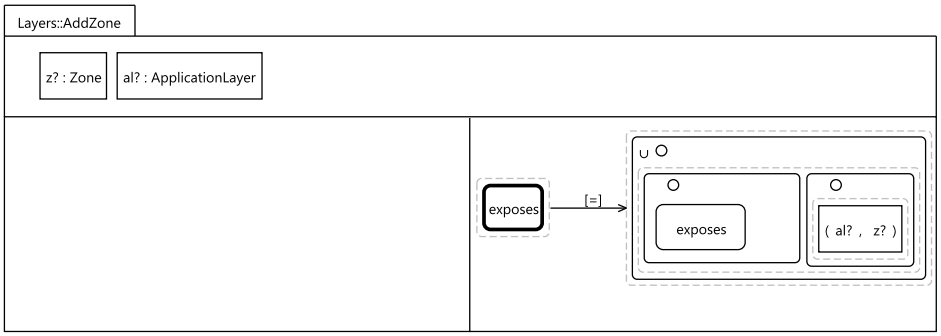


Figure A.53: The `Layers' AddZone` operation

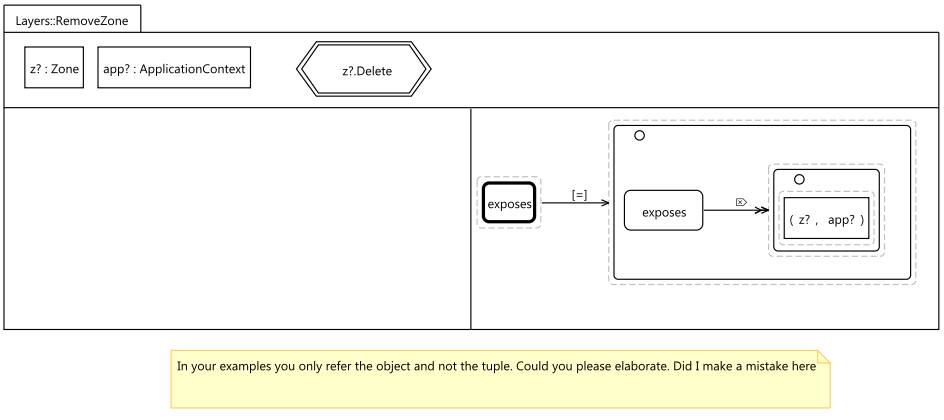


Figure A.54: The `Layers' RemoveZone` operation

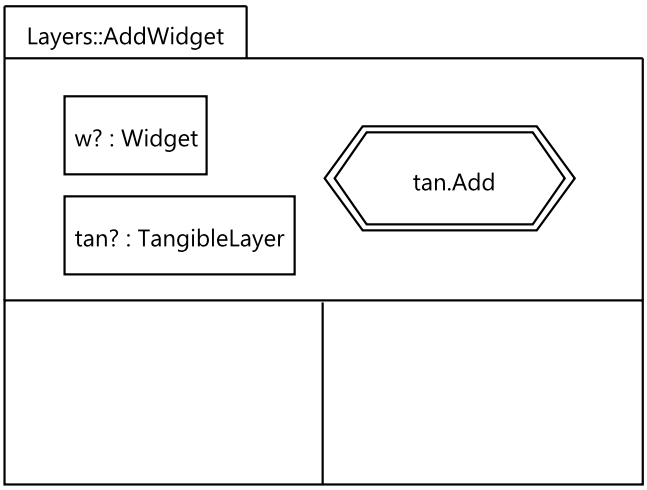


Figure A.55: The `Layers' AddWidget` operation

A.9 The Widgets package

The *`Widgets`* package contains all principles specified by Bicheler; a `Widget` has some `Attributes` and one identifier expressed through the `has` and `id` parameter edges respectively. Furthermore, it states that a `Widget's Identity` is given by the referenced `Functions`. Lastly, the SD shows that a `Widget` is composed of at least one `WidgetComponent` imported from the *`EndPoints`* package.

The requirement to offer a default `Widget` with a default `Function`, allowing for some default behaviour is enforced by the respective default operations. The *`Widget`* package also offers the global `AddAction` operation. This operation is part of the behaviour specification to create `Functions` and `Mappings`. As mentioned previously, the behaviour is only partially implemented to show the feasibility and not get caught up with repetitive and time intensive modelling tasks.

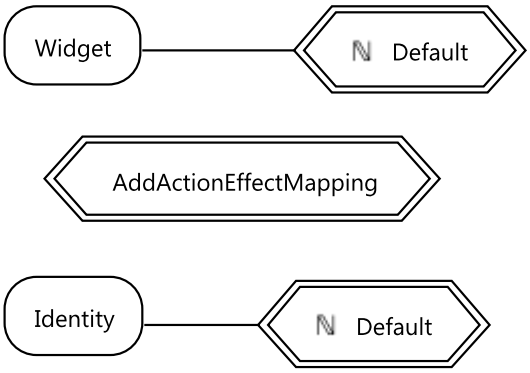


Figure A.56: The *Widgets'* package diagram

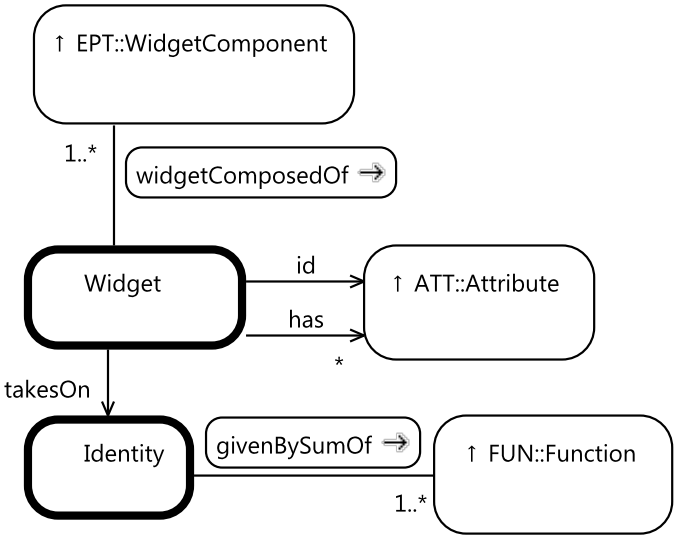


Figure A.57: The *Widgets'* structure diagram

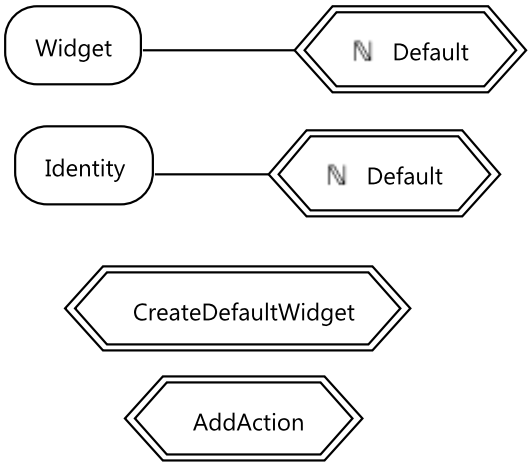


Figure A.58: The *Widgets'* behaviour diagram

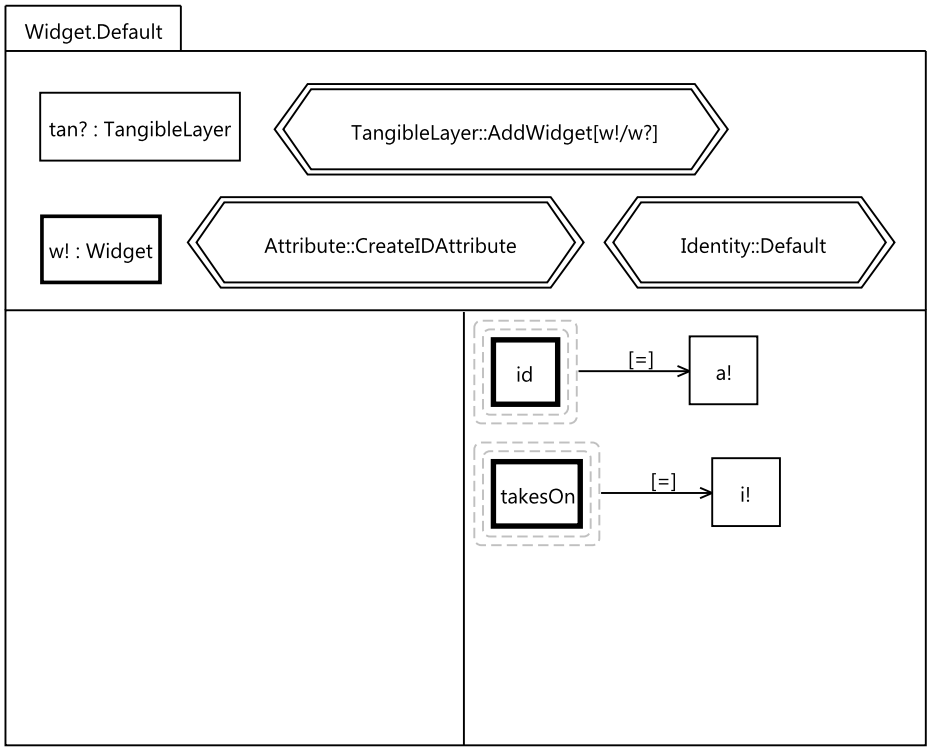


Figure A.59: The `Widget's Default` operation

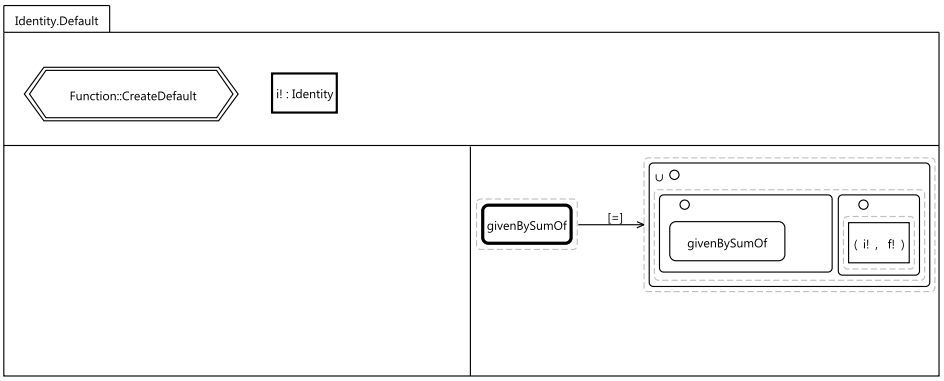


Figure A.60: The `Identity's Default` operation

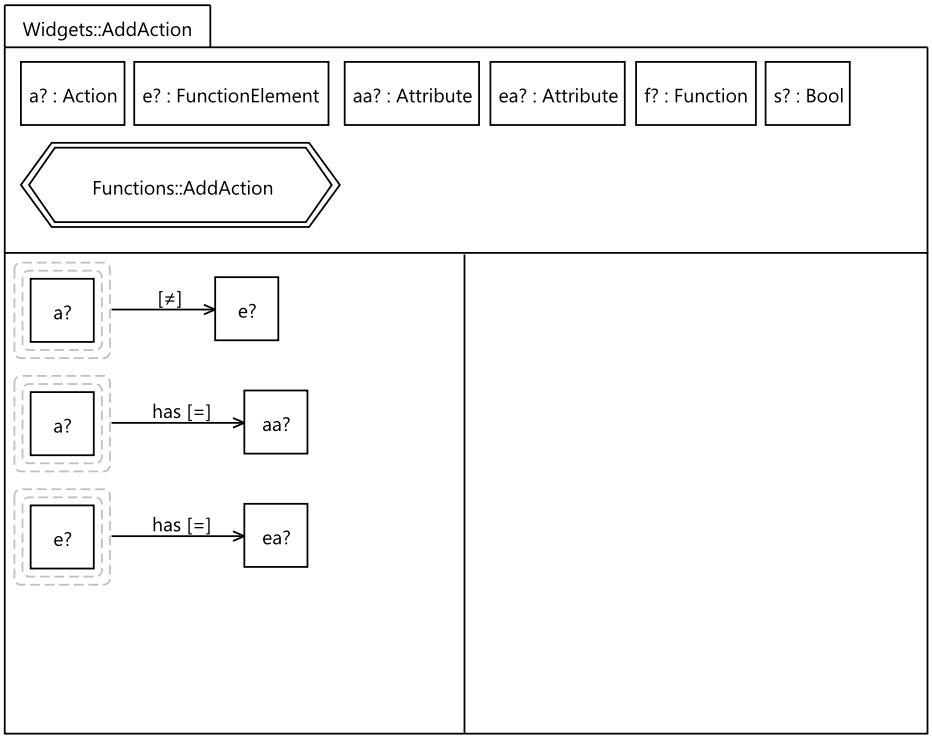


Figure A.61: The `Widget's AddAction` operation

A.10 The TUI package

The *TUI* package merges all previously defined concepts together. The package consists only of a PD A.62, exposing an ensemble package for the TUI that creates a global state space for all TUI concepts.

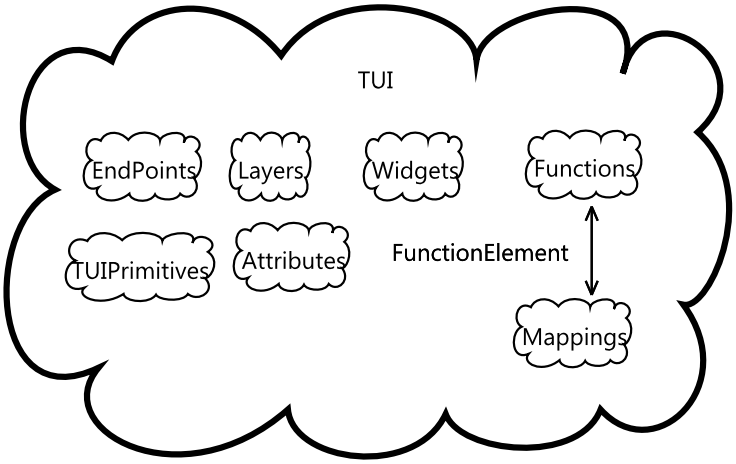


Figure A.62: The *TUI's* package diagram

Appendix B

BPMN2 metamodel VCL packages

The following sections present all Visual Contract Language (VCL) packages needed to build the Business Processes Model and Notation, version 2 (BPMN2) metamodel as given by its specification document [23]. The packages will detail all diagrams. Each section will provide an overview of the package and complement the general description given in Section 5.5.

For an introduction on VCL, please consider reading Section 4. In this appendix, the following notation is used; Structural Diagrams (SD), Behavioural Diagrams (BD), Package Diagrams (PD), Assertion Diagrams (AD), Contract Diagrams (CD). Table B.1 lists all VCL packages and the corresponding figures for each VCL package.

Table B.1: BPMN2 metamodel VCL packages and figures

Package name	Figures
Pimitives B.1	PD B.1 SD B.2
Extensibilities B.2	PD B.3 SD B.4 BD B.5 Extension_Default CD B.6
BaseElements B.3	PD B.7 SD B.8 BD B.9 Documentation_Default CD B.10
Foundation B.4	PD B.11

Continued on next page

Table B.1 – *Continued from previous page*

Package name	Figures
Infrastructures B.5	PD B.12 SD B.13 BD B.14 Definitions_Default CD B.15
ItemDefinitions B.6	PD B.16 SD B.17 BD B.18 ItemDefinition_Default CD B.19
Messages B.7	PD B.20 SD B.21
Artifacts B.8	PD B.22 SD B.23 BD B.24 Association_Default CD B.25 TextAnnotation_Default CD B.26
Resources B.9	PD B.27 SD B.28
ResourceAssignments B.10	PD B.29 SD B.30 ResourceRole_BindingOnReference AD B.31
Expressions B.11	PD B.32 SD B.33
Errors B.12	PD B.34 SD B.35
Collaborations B.13	PD B.36 SD B.37 BD B.38 GetCollaborationGivenParticipant AD B.41 MessageFlowSourceConstraints AD B.39 MessageFlowTargetConstraints AD B.40
GlobalTasks B.14	PD B.42 SD B.43
Services B.15	PD B.44 SD B.45
ItemAwareElements B.16	PD B.46 SD B.47
IOSpecifications B.17	PD B.48 SD B.49 InputSetReferenceIntegrity AD B.52 OutputSetReferenceIntegrity AD B.53

Continued on next page

Table B.1 – *Continued from previous page*

Package name	Figures
	DataInput_Default CD B.50 DataOutput_Default CD B.51
DataAssociations B.18	PD B.54 SD B.55 SingleSource AD B.56 AssignmentNotNull AD B.57 DefinitionsMatch AD B.58 ReferenceIntegrity AD B.59
Data B.19	PD B.60 SD B.61 InputCollectionTypeMatches AD B.62
Processes B.20	PD B.63 SD B.64 BD B.65 Process_Default CD B.66
Lanes B.21	PD B.67 SD B.68
Escalations B.22	PD B.69 SD B.70
EventDefinitions B.23	PD B.71 SD B.72 BD B.73 GetValidSubProcessStarts AD B.74
Events B.24	PD B.75 SD B.76 ConsistencyRequirement AD B.78 InterruptingBehaviour AD B.77
FlowElements B.25	PD B.79 SD B.80 BD B.81 SourceBoundaryEventConstraints AD B.83 SourceIntermediateEventConstraints AD B.84 SourceStartEndEventConstraints AD B.82 TargetBoundaryEventConstraints AD B.86 TargetIntermediateEventConstraints AD B.87 TargetStartEndEventConstraints AD B.85 EventSubProcessUnconnected AD B.88 GetCollaborationGivenFlowNode AD B.89 MessageFlowsSpanPools AD B.90 SequenceFlowAttributeConstraints AD B.93 SequenceFlowSourceConstraints AD B.91

Continued on next page

Table B.1 – *Continued from previous page*

Package name	Figures
	SequenceFlowTargetConstraints AD B.92
FlowElementContainers B.26	PD B.94 SD B.95 BD B.96 ExecutedImmediately AD B.97 GetProcessGivenFlowElement AD B.98
CallableElements B.27	PD B.99 SD B.100
Common B.28	PD B.101
LoopCharacteristics B.29	PD B.102 SD B.103 MILCharacteristics_Instantiation AD B.104
SubProcesses B.30	PD B.105 SD B.106 OneStartEvent AD B.107 AdHocSubProcessRestrictions AD B.109 Transaction_SpecificMethods AD B.108
Activities B.31	PD B.110 SD B.111 BD B.112 Activity_FrameConditions AD B.113 IOConstraints AD B.114 Activity_Default CD B.115
Gateways B.32	PD B.116 SD B.117 BD B.118 ConvergingGatewaysBundle AD B.120 DivergingGatewaysDiffuse AD B.121 EventBasedGatewayConstraints AD B.119 MixedGatewayConstraints AD B.122 EventBasedGateway_Default CD B.123
Tasks B.33	PD B.124 SD B.125 BD B.126 ServiceTaskMessageConstraint AD B.127 BusinessRuleTask_Default CD B.128 ReceiveTask_Default CD B.131 SendTask_Default CD B.130 ServiceTask_Default CD B.129 UserTask_Default CD B.132

Continued on next page

Table B.1 – *Continued from previous page*

Package name	Figures
Participants B.34	PD B.133 SD B.134 BD B.135 GetParticipantGivenProcess AD B.136
Core B.35	PD B.137

B.1 The Primitives package

The *Primitives* package includes primitives from MOF not defined as primitives in VCL such as `String` or `Bool`, and explicitly defines sets that are implicit in the PBMN2 specification such as `ItemKind` or `GatewayDirection`. `Element` is the topmost enclosing set. It is no definition set and as such is not restricted to include only the modelled enclosed sets. This design choice has been made in order to use `Element` as an analogy to the `MOFElement` that is sometimes used by the BPMN2 specification to not restrict the specification to much and allow for future extensions.

Due to VCL requiring set names to be unique, some concepts had to be renamed and therefore do not figure as such in the specification. An example is `GatewayDirection` and `AssociationDirection` which only appear as `Direction` in the BPMN2 document. The package also contains concepts that are not included as such in BPMN2, for example, `UID`. While the specification mentions and requires the uniqueness of identifiers at some points, it does not mention how it is enforced. As VCL lacks universal quantifiers, a design decision was taken to simply include `UID` as a primitive.

Figure B.1: *Primitives'* package diagram

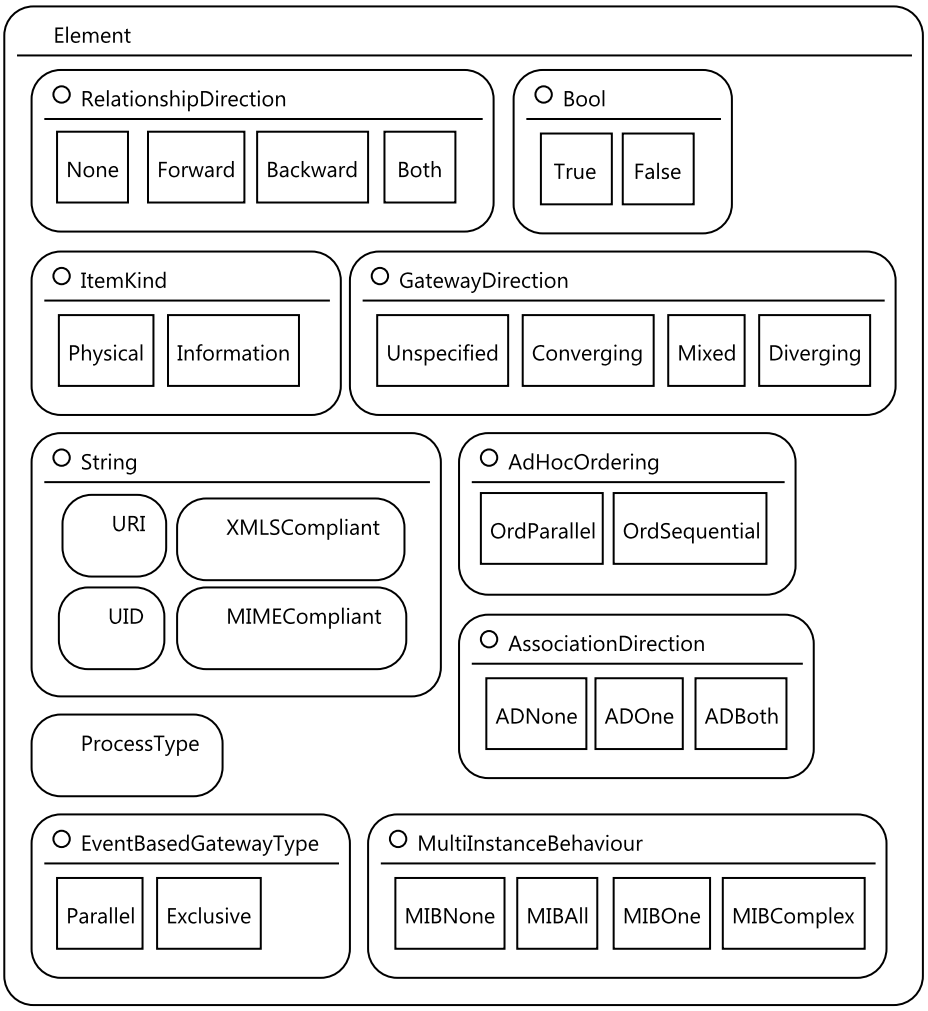


Figure B.2: *Primitives'* structural diagram

B.2 The Extensibilities package

The package defines `Extension`, an `ExtensionDefinition` and its attributes by giving their `ExtensionAttributeDefinition` and `ExtensionAttribute-Value`. The `Extension's Default` constructor satisfies the requirement for default values.

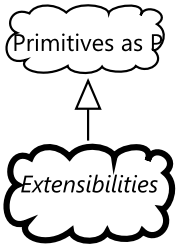


Figure B.3: The *Extensibilities'* package diagram

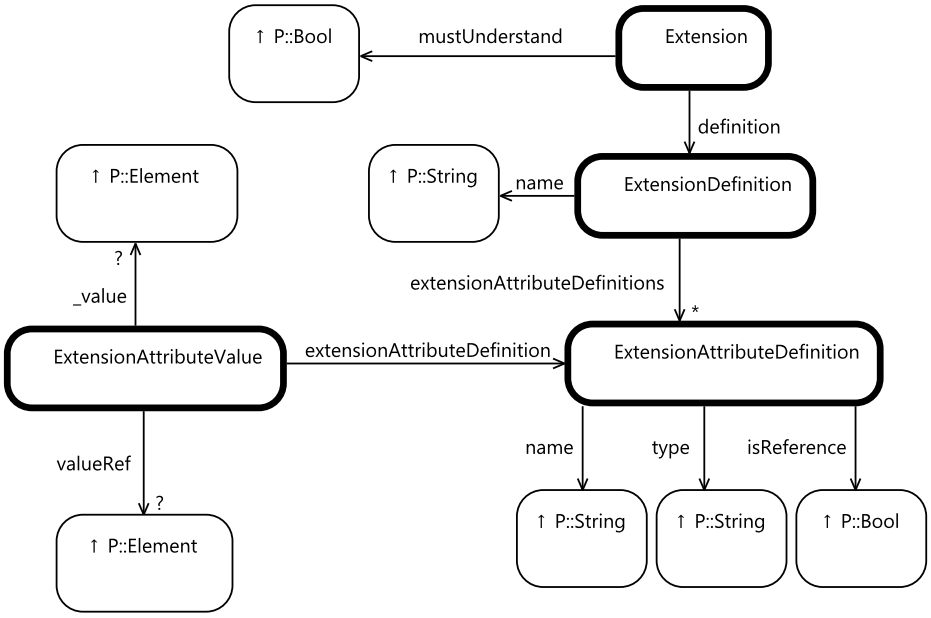


Figure B.4: The *Extensibilities'* structure diagram

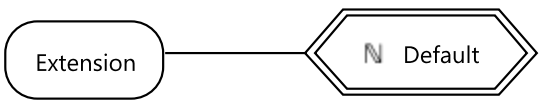


Figure B.5: The *Extensibilities'* behaviour diagram

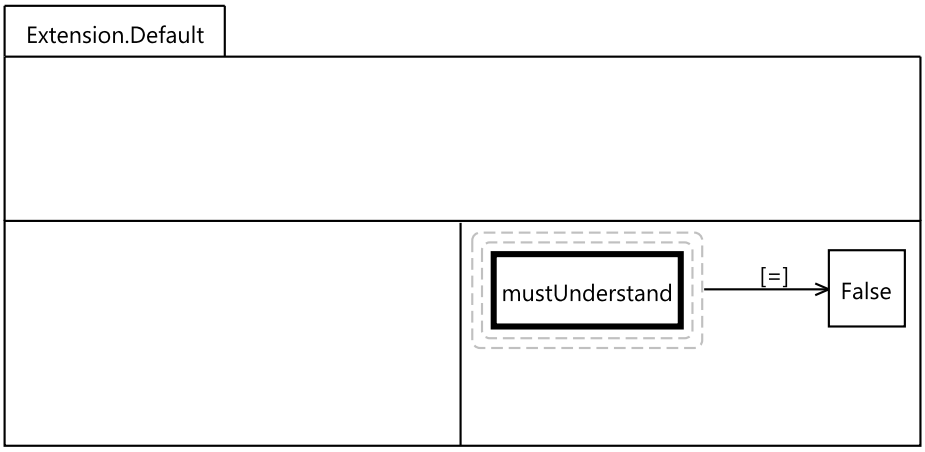


Figure B.6: The `Extension's Default` operation, setting default values

B.3 The BaseElements package

The *`BaseElements`* package contains `BaseElements`, the topmost abstract class in BPMN2 for most model elements. In VCL, this is expressed by `BaseElement` encompassing all elements and marking it with a ◯ symbol next to its name, meaning that `BaseElement` is defined through its components, not ever existing outside of a scope also defined by one of its subsets. While this does not convey the same semantics as the UML2's abstract class, it is as close as modelling in VCL can get.

The specification also introduces the concept of `Documentation` and some ambiguity. On figure 8.5 on page 55 in the specification document, the end point cardinality for the composition relation between `BaseElements` and `Documentation` is set to *. The textual description on page 56 on the Document class however mentions,

> "*All BPMN elements that inherit from the* ***`BaseElements`*** *will have the capability, through the* ***`Documentation`*** *element, to have one (1) or more text descriptions of that element.*"

This contradicts the class diagram. While it is always a good practice to have a documentation, forcing all elements to have an accompanying description is cumbersome and might not be in the best interest of the language regarding ease of use. Therefore, the cardinality has been maintained at *. Furthermore, the specification document shows `BaseElement` aggregate `Documentation`, one of its child elements. This violates the paradigm of

separation of concerns. Therefore, another superposing element, `BaseElementContainer` was introduced in the VCL diagram. It includes `Documentation` as well as `BaseElement`.

`Relationship` is intended to allow BPMN2 `Elements`, or `MOFElements` to be more specific, to relate to non-BPMN2 elements. `RootElement` is a concept that allows BPMN2 elements that are defined in the scope of `Definitions` to persist beyond the life-cycle of their parent elements. In addition to defining all those concepts, the *`BaseElements`* VCL package also defines a local `Default` operation for `Documentation`, setting the default value as required by the specification.

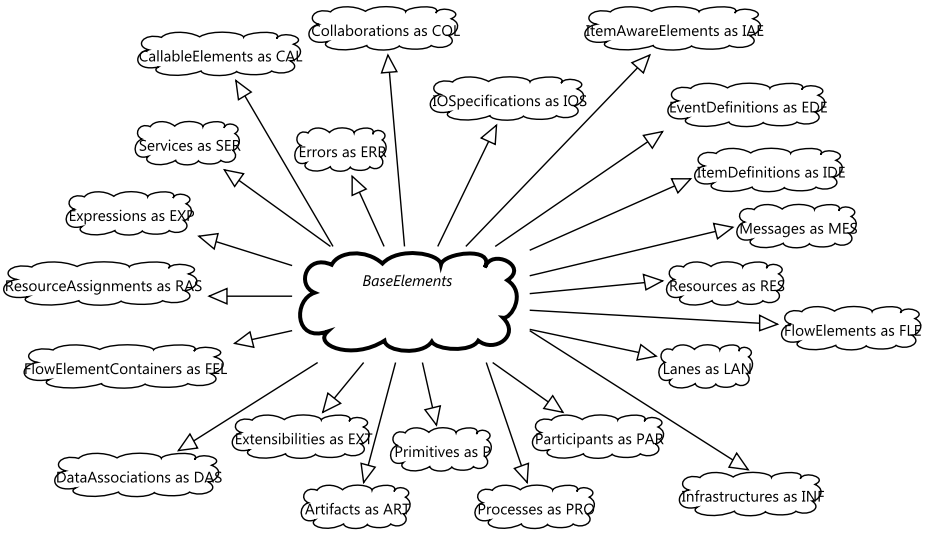


Figure B.7: The *`BaseElements'`* package diagram

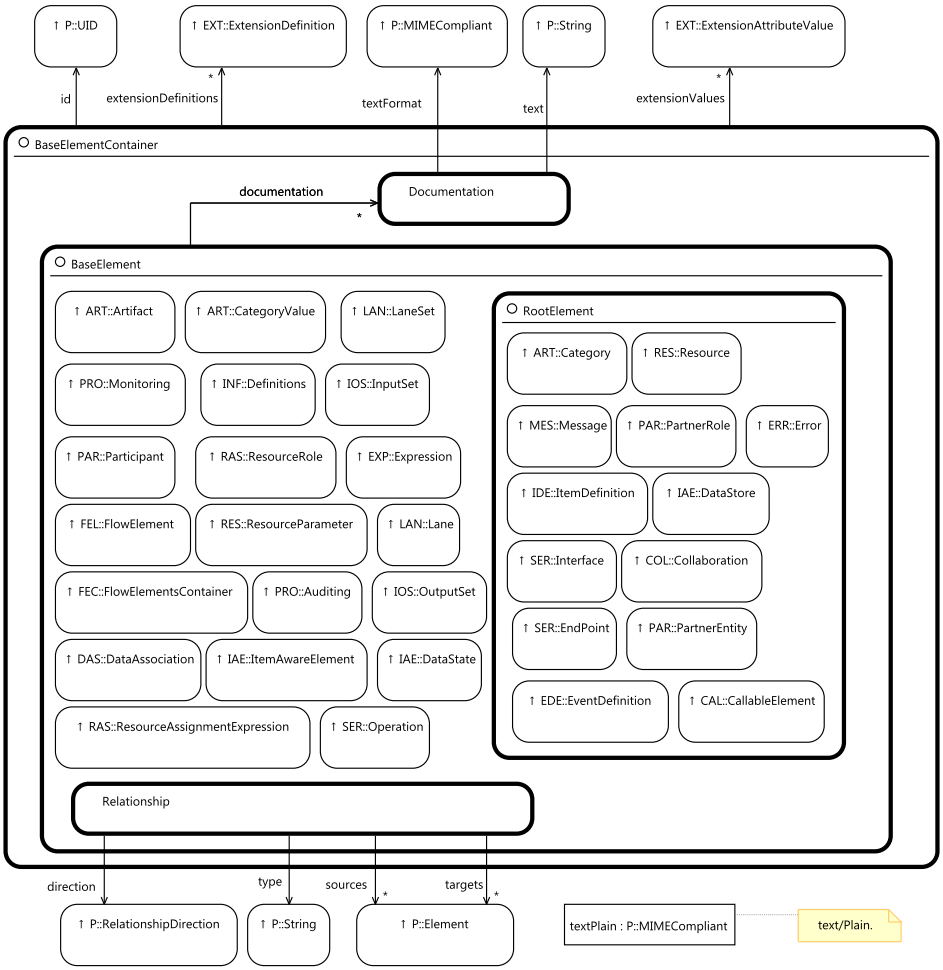


Figure B.8: The *BaseElements'* structure diagram

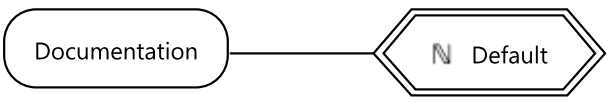


Figure B.9: The *BaseElements'* behaviour diagram

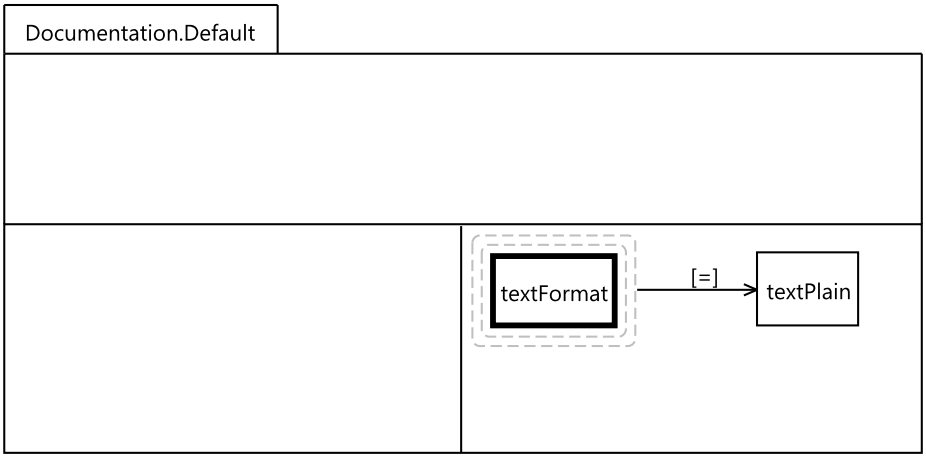


Figure B.10: The `Documentation's Default` operation, setting default values

B.4 The Foundation package

The *`Foundation`* VCL ensemble package groups all concerns from the *`BaseElements`* and *`Extensibilities`* VCL packages.

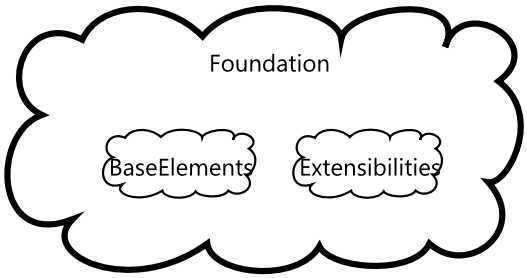


Figure B.11: The *`Foundation's`* package diagram

B.5 The Infrastructures package

The *Infrastructures* package contains two elements, `Definitions` and `Import`. Other than defining the namespace, `Definitions` also sets the expressions and type language for contained elements. In order to assure an independent life-cycle, `Definitions` hold a list to all rooted `RootElements`. On the VCL SD for `Definition`, rather than referencing `RootElement`, and `Relationship` for that matter, the SD refers to the concepts directly. This is due to the high coupling of concepts in BPMN2 and VCL not accepting circular imports. Therefore, the concepts are merged in the *Core* VCL package.

Moreover, `Definitions` holds a reference to the `Import` making external elements available and a reference, by name only, to the exporter, and exporter version, which is exporting the BPMN2 file. The implementation deviates slightly from the specification as that it does not mention the BPMNDI which has been omitted to simplify the modelling. The package also defines a `Default` operation local to `Definition` setting default values as required by the specification.

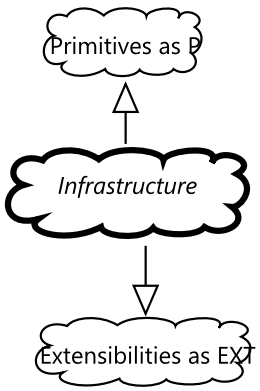


Figure B.12: *Infrastructures'* package diagram

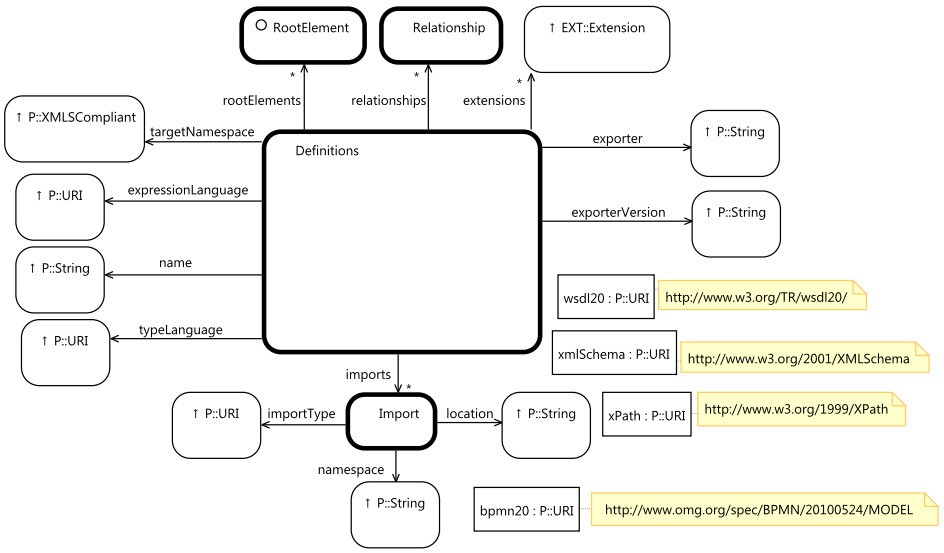


Figure B.13: *Infrastructures'* structural diagram

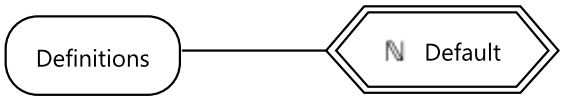


Figure B.14: *Infrastructures'* behaviour diagram

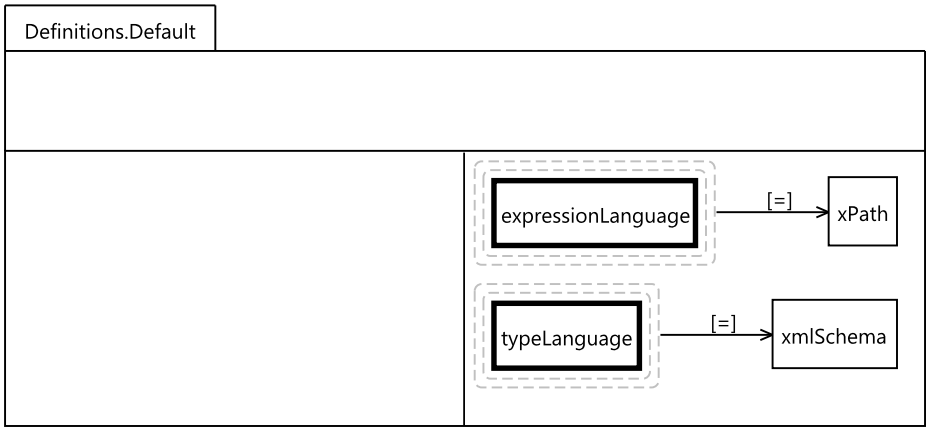


Figure B.15: The `Definitions' Default` operation, setting default values

B.6 The ItemDefinitions package

The concept of `ItemDefinitions` is used to separate the structural definitions from model elements that are exchanged during a `Process`, thereby decoupling definition from use. The structural definition may be imported in which case an `Import` is held in reference. By the `ItemKind`, items are separated into either `Physical` or `Information` items. The actual structure, which might also represent a collection, is given by a referenced structure `Element`. The specification mentions default values which are set by the `IdemDefinition's Default` operation.

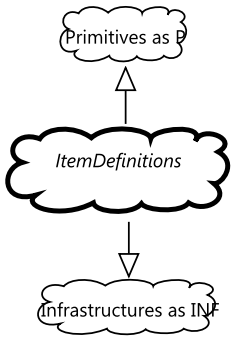


Figure B.16: The *ItemDefinitions'* package diagram

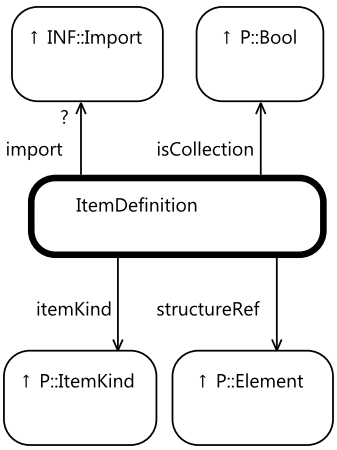


Figure B.17: The *ItemDefinitions'* structural diagram

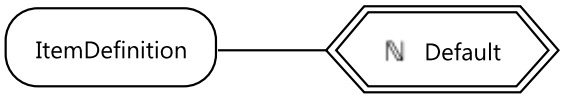


Figure B.18: The *ItemDefinitions'* behaviour diagram

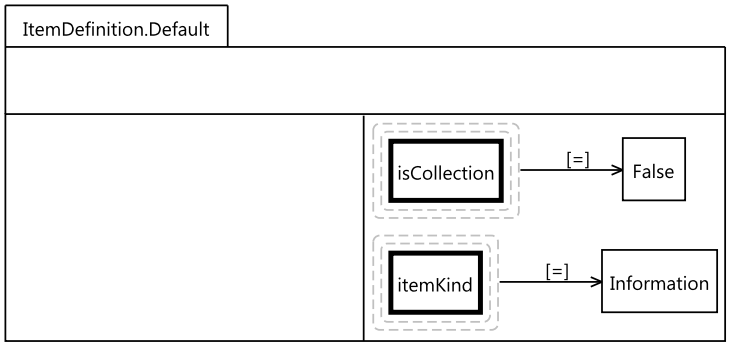


Figure B.19: The `ItemDefinition's Default` operation

B.7 The Messages package

The *Messages* VCL package implements the concept of information exchange between `Participants`. While constraints apply to the sending and receiving of `Messages` all those constraints are expressed on the corresponding `FlowElements`. A `Messages` references an `ItemDefinitions`, defining its payload and a textual description to make it easily identifiable.

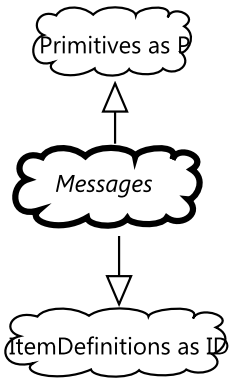


Figure B.20: The *Messages'* package diagram

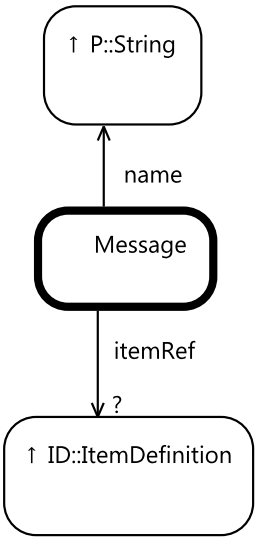


Figure B.21: The *Messages'* structural diagram

B.8 The Artifacts package

Subsetting in VCL is expressed through insideness, hence `TextAnnotation`, `Association`, and `Group` are contained within the `Artifact` blob. All `Artifact` subsets must satisfy the constraints on sequence flow connections as defined by the specification document. These constraints are detailed by assertions and invariants in the *SequenceFlow* VCL package.

An `Association` has two uses. It either binds an `Artifact` to a `FlowElements` or it marks `Activities` used for compensation. An `Association` has a direction as defined by the *Primitives'* `AssociationDirection` accessed by a reference, requiring the import of said package. Furthermore, with the `Association` being a link, it evidently needs to reference a source and a target. This is done by the appropriately named relational edges which end in a referenced `BaseElements`. To satisfy the call for default values, `Association` features a `Default` operation.

`Group` offers a visual way of representing `Categories`. They visually enclose `FlowNodes` and thereby group them. `Groups` do not interfere in the flow of the model. `Groups` are tied to a `CategoryValue` which, by its value, identifies the `Group`. A user defined named `Category` can have multiple `CategoryValues` associated to it, aggregating different `Groups` into once larger concept.

`TextAnnotation` implements the concept of textual user annotation. It does not influence the flow of the `Process`. The `TextAnnotation` can be connected by an `Association` to any `BaseElements`. The body of the `TextAnnotation`, its text is specified to be a `String`. The requirement for the text to be `MIMECompliant` is satisfied by typing. The `TextAnnotation's Default` operation, satisfies the need for default values.

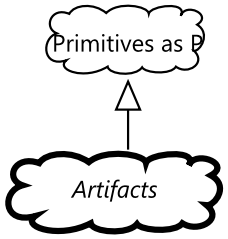


Figure B.22: The *`Artifact's`* package diagram

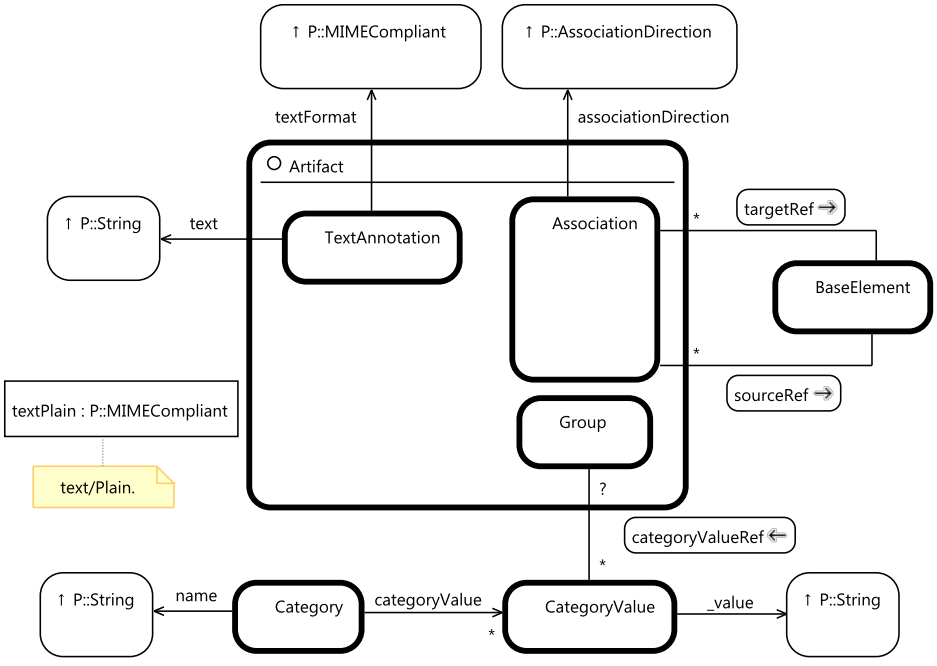


Figure B.23: The *`Artifact's`* structural diagram

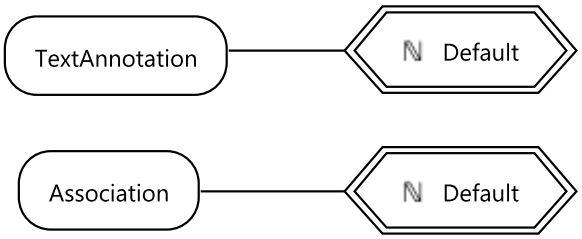


Figure B.24: The *Artifact's* behaviour diagram

Figure B.25: The *Association's* `Default` operation

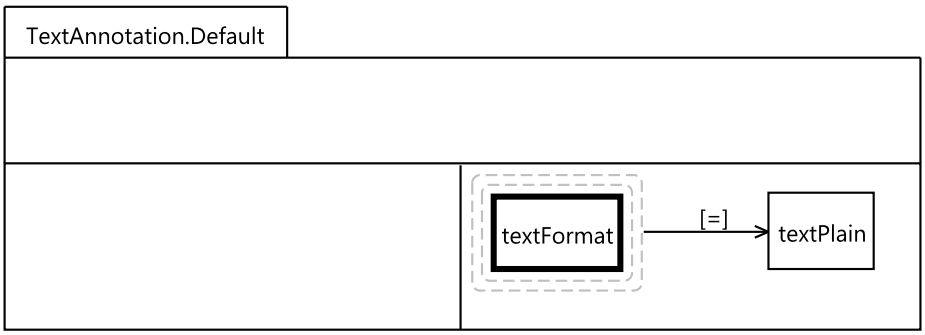


Figure B.26: The *TextAnnotation'* `Default` operation

B.9 The Resources package

The *Resources* package implements the concept of resources as commonly used in business contexts. A named `Resources` is defined by the optional `ResourcesParameters` it may have. The parameters are typed by and named by `ItemDefinitions`.

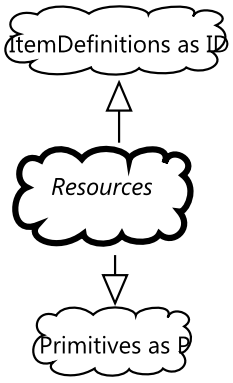


Figure B.27: The *Resources'* package diagram

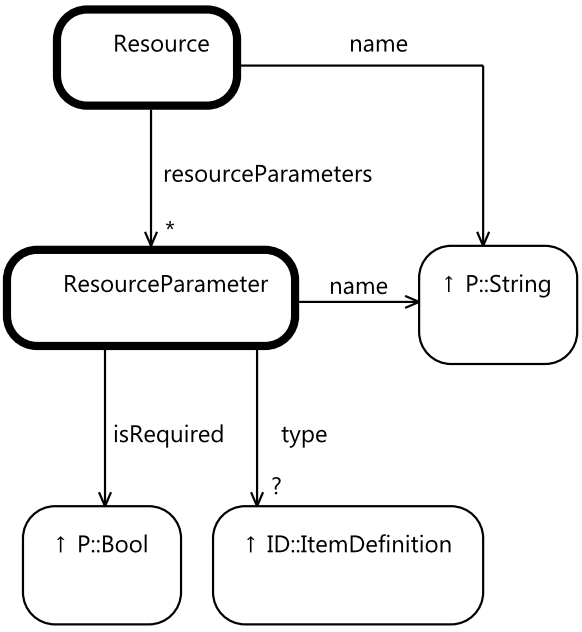


Figure B.28: The *Resources'* structural diagram

B.10 The ResourceAssignments package

A `ResourceRole` includes recursive subsets which refine the concept by, in the order of insideness, `Performer`, `HumanPerformer` and lastly `PotentialOwner`. These roles are used for `Resources` assigned to `Activities`. `ResourceRole` either references a `Resource` directly or uses a `ResourceAssignmentExpression` to be address a `Resource`. This expression should return elements of type `Resource` only. However, the necessary facility to express the return type is only given for `FormalExpression`. It is unknown if the intention of the specification was to imply the use of a `FormalExpression` or if the reference to the underspecified `Expression` was an oversight. Hence, the requirement has not been modelled. `ResourceParameterBindings` are used to bind `Resources` referenced by the `ResourceRole's resourceRef` property to an actual `ResourceParameter`, a subset of all parameters as defined by the `Resource` that are used in the scope of its assignment.

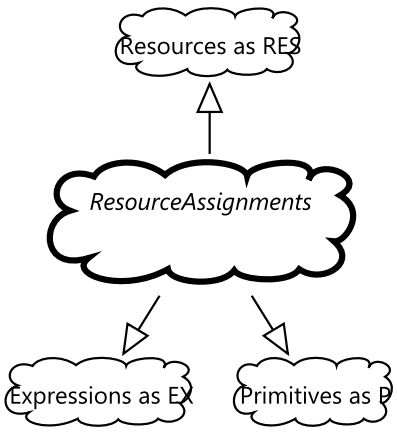


Figure B.29: The *`ResourceAssignments'`* package diagram

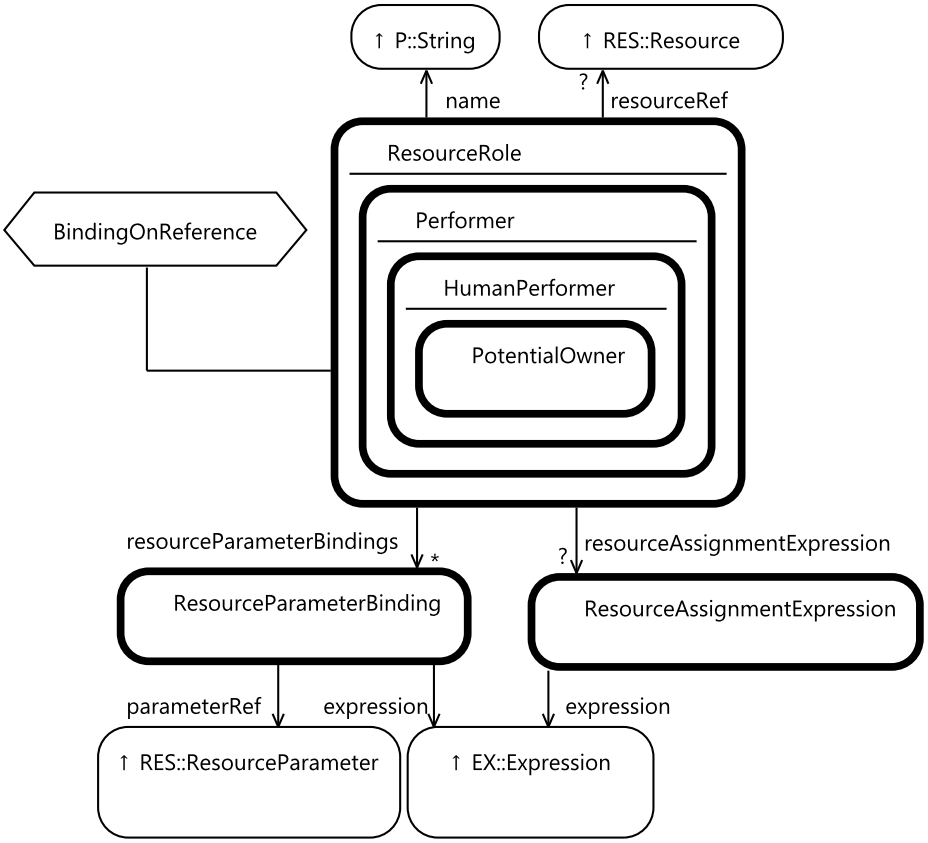


Figure B.30: The *ResourceAssignments'* structural diagram

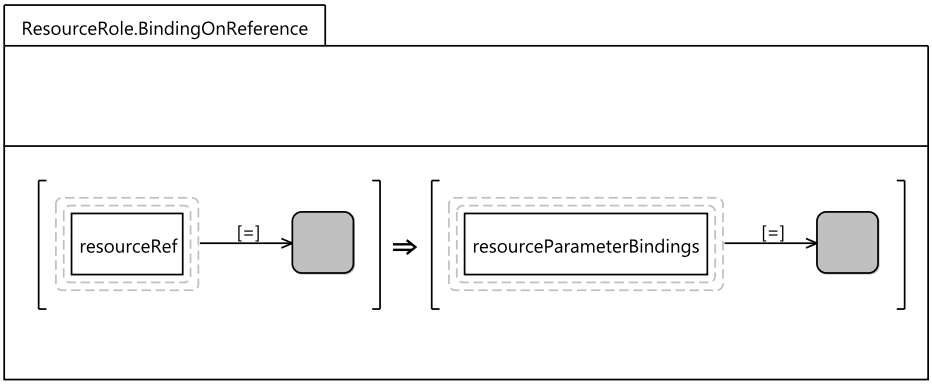


Figure B.31: The `BindingOnReference` assertion expressing constraints on the `ResourceRole`

B.11 The Expressions package

`Expressions` are used to specify natural language constructs. The `Formal-Expressions` concept is a subset of `Expressions`, catering only to formal language constructs. The `FormalExpressions` references an `ItemDefinitions`, defining the return type. The `langauge` property overrides the `Definition's expressionLanguage` property allowing for a local specification and alternation of the language in use. The requirement for the language property to be given as a `URI` is fulfilled by typing. The `body` property specifies the contents of the `FormalExpressions` by referencing an `Element`, enabling the expression of mixed language constructs. The statement by the specification that this property is not relevant when a XML Schema is used for interchange is not modelled as it is deemed to be a choice external to the model.

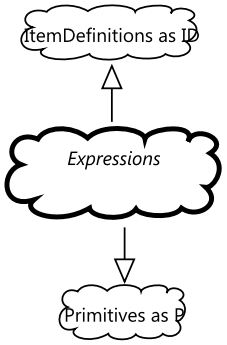


Figure B.32: The *Expressions'* package diagram

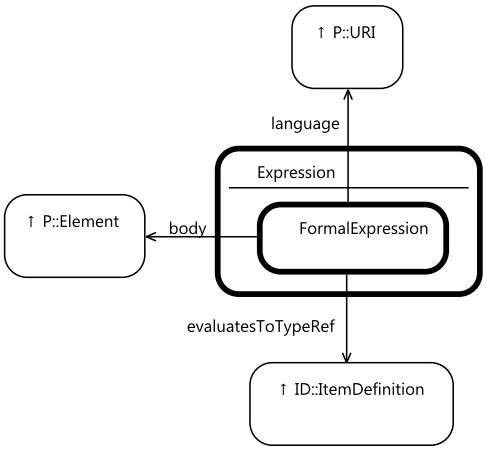


Figure B.33: The *Expressions'* structural diagram

B.12 The Errors package

The `Error's` structure is specified by an `ItemDefinition` and has a `name` as well as an `errorCode`. The specification mentions constraints on the `errorCode`, stating that for certain `Event` subsets, an `errorCode` must or may be specified. However, the class diagram of the specification document on page 81, does shows that the `Error` has the non-optional `errorCode` property. This is expressed in VCL as well and therefore, an `errorCode` must be supplied for all `Errors`. This satisfies the requirement but may, due to the ambiguity of the specification document, not model the intent correctly.

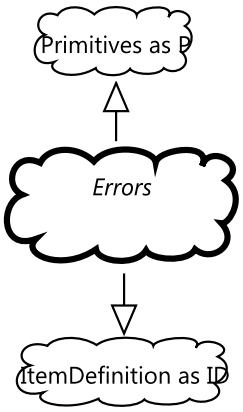


Figure B.34: The *Errors'* package diagram

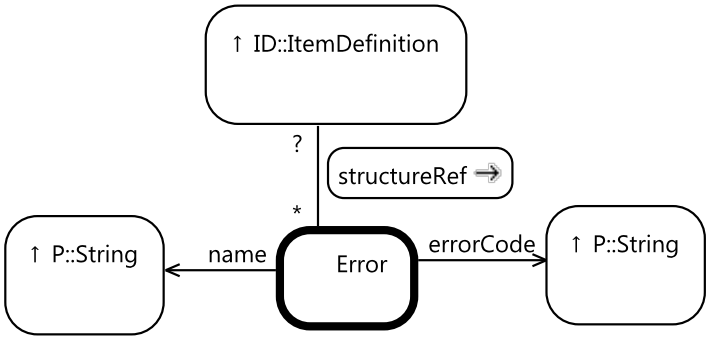


Figure B.35: The *Errors'* structural diagram

B.13 The Collaborations package

The *Collaborations* VCL package groups all concepts surrounding `Collaborations`. Other than a `name`, a `Collaborations` includes;

- an `artifact` reference listing all `Artifacts` contained within the `Collaborations`,
- a `participants` reference listing all contained `Participants`,
- a list of all `Participants` contained in another `Collaborations` trough the use of a `ParticipantAssociation`,
- a `messageFlows` reference listing all contained `MessageFlows`,
- a boolean property `isClosed` specifying that, unless false, only `MessageFlows` can carry `Messages` between `Participants`.

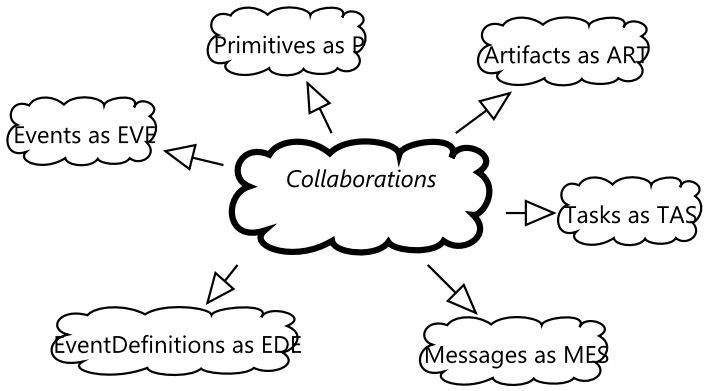


Figure B.36: The *Collaborations'* package diagram

This package also introduces the concept of `InteractionNode` which is defined as a general concept for `Participants`, `Events` and `Tasks`, providing a single element to serve as source and target of `MessageFlows`. In the VCL SD this is expressed through insideness and `InteractionNode` being marked as definition of its subsets. `ParticipantsAssociation` is a concept is mainly used for scenarios not covered in the scope of this document. Its sole use is therefore to specify a mapping between two `Participants` contained in different BPMN2 diagrams.

`MessageFlows` are used to symbolise the flow of `Messages` between `InteractionNodes`. `MessageFlows` are given a `name` and, optionally, refer to the exchanged `Message`. The invariant on `MessageFlows`, `MessageFlowsSpanPools` is expressed in the *FlowElements* package in Section B.25.

The specification document imposes certain requirements regarding `MessageFlows` and the `InteractionNodes` that may be their source, respectively target. The `MessageFlowSourceConstraints` and `MessageFlowTargetConstraints` invariants satisfy these requirements by stating that, `EndEvents` and `IntermediateThrowEvents` throwing a `Message` cannot be the target of `Messages` themselves. In an inverse argument, `StartEvents` and `IntermediateCatchEvents` catching a `Message` cannot be the source of `Messages`.

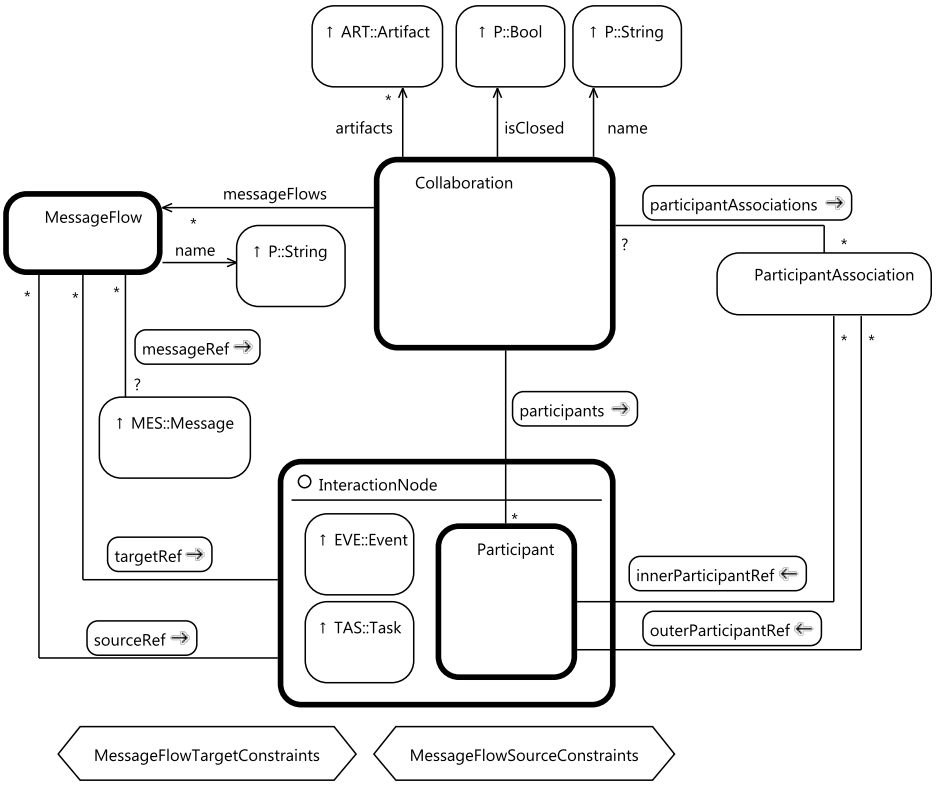


Figure B.37: The *Collaborations'* structural diagram

Figure B.38: The *Collaborations'* behaviour diagram

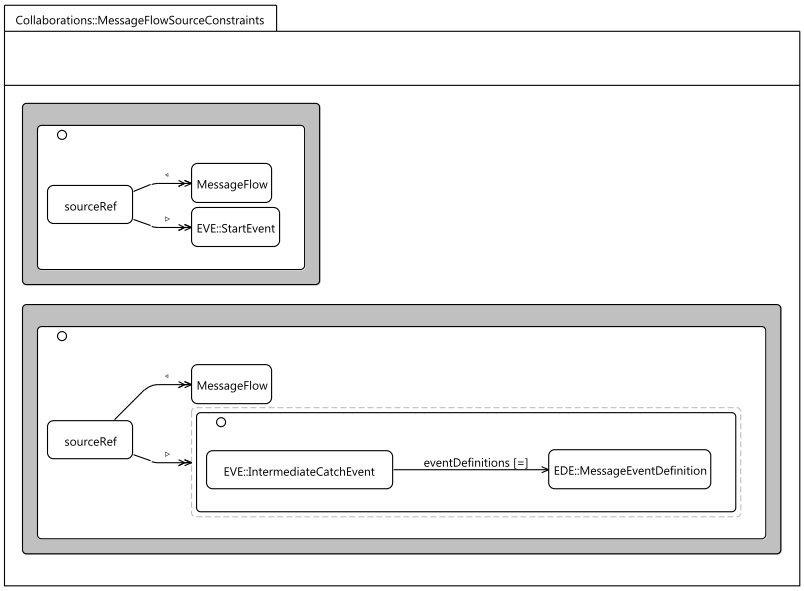


Figure B.39: The `MessageFlowSourceConstraints` assertion expressing constraints on the `MessageFlow`

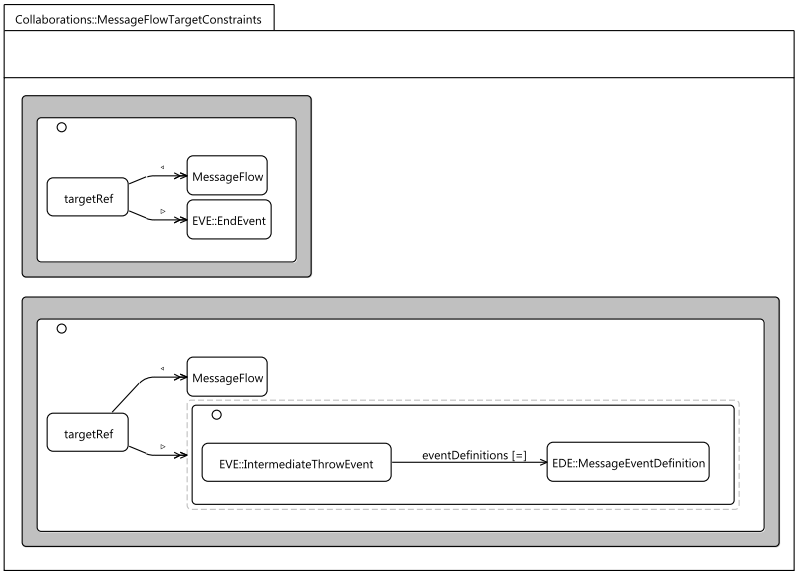


Figure B.40: The `MessageFlowTargetConstraints` assertion expressing constraints on the `MessageFlow`

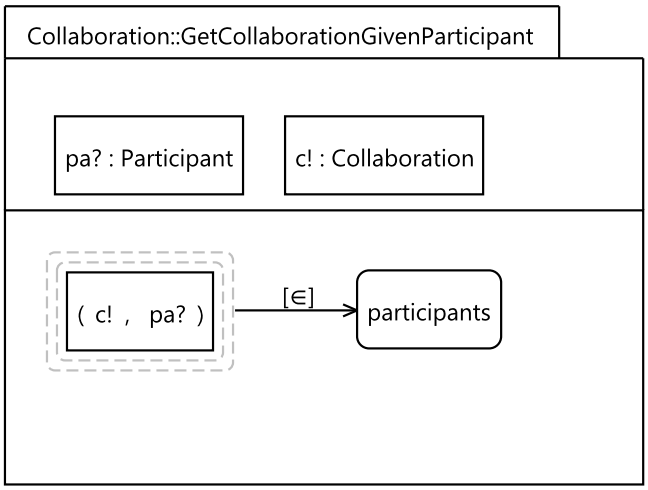


Figure B.41: The *Collaborations'* `GetCollaborationGivenParticipant` operation

B.14 The GlobalTasks package

The BPMN2 specification mentions that `GlobalTask` types are only a subset of the `Task's` types. This is expressed through insideness in VCL. Therefore, `GlobalUserTask`, `GlobalBusinessRuleTask`, `GlobalManualTask` and `GlobalScriptTask` are defined. The `resources` property provides a link to all `Resources` the `GlobalTask` is linked to. The idea behind `GlobalTask` is to provide a `Task` that is accessible and reusable across `Collaborations` and that symbolises a synchronised execution of a given `GlobalTask`.

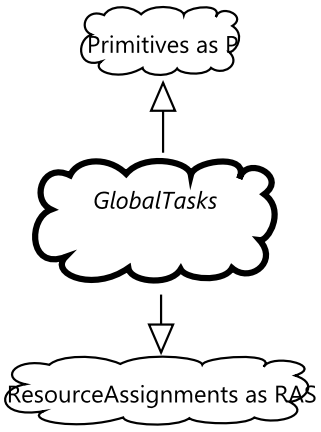


Figure B.42: The *GlobalTasks'* package diagram

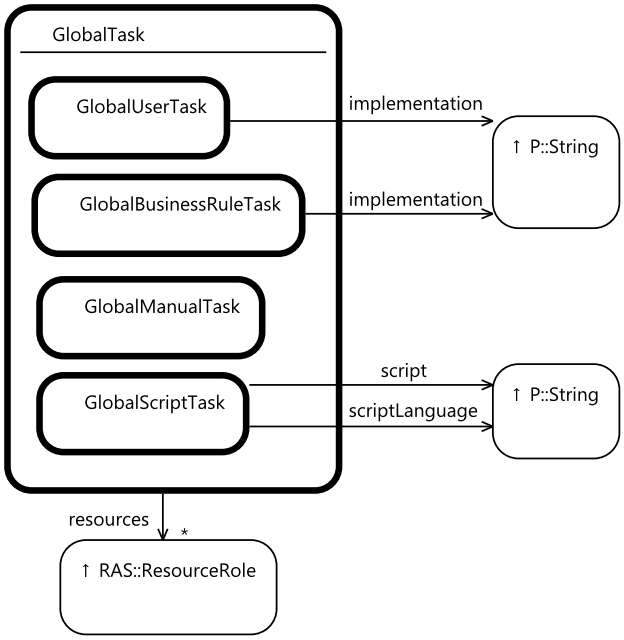


Figure B.43: The *GlobalTasks'* structural diagram

B.15 The Services package

`Operations` are distinguished by their `name`. The `implementationRef` property refers to a concrete implementation technology used to express the `Interface`. Most of the specification of *Services* is outside of the scope of the specification document and, hence, the scope of this document as well. *Service's* `EndPoints` specify the address at which a *Service* can be accessed. `Operations` specify, like`Interfaces`, an `implementationRef` and a `name`. Moreover, they specify what `Messages` can be received and produced by a *Service* as well as detailing any `Errors` may be returned.

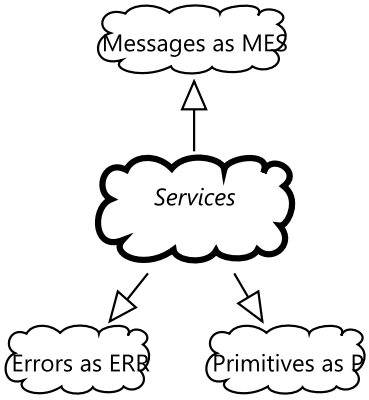


Figure B.44: The *Services'* package diagram

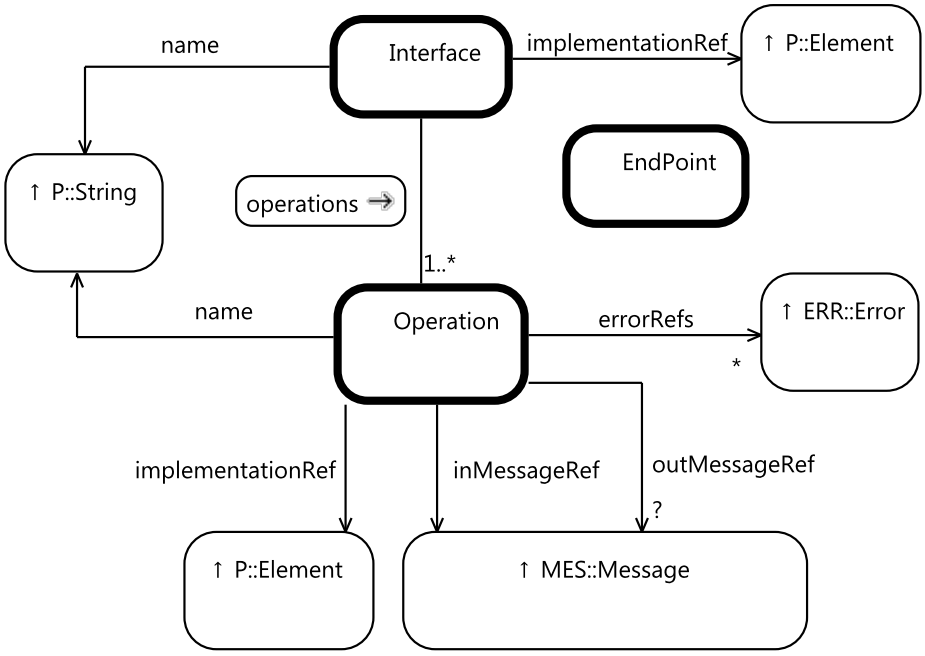


Figure B.45: The *Services'* structural diagram

B.16 The ItemAwareElements package

The `itemSubjectRef` property provides a reference to the variable's structure. Every variable has a state modelled by the `DataState`. There are seven subsets which define `ItemAwareElement`. This is shown by insideness in VCL. `DataObjects` represent data that is contained within a `Process'` or `SubProcess'` flow. To reuse `DataObjects`, `DataObjectReferences` can be used. To model non-volatile data, `DataStores` are used. They allow data to be retrieved and stored beyond the scope of the `Process` or `SubProcess`. They can be referenced using `DataStoreReferences`. `Properties` enrich `Processes`, `Activities`, or `Events` by providing a mean to attribute additional, visualised data to those BPMN2 `FlowElements`. Data that is produced and communicated during an `Activity` or `Process` execution is modelled by `DataInput` and `DataOutput` respectively.

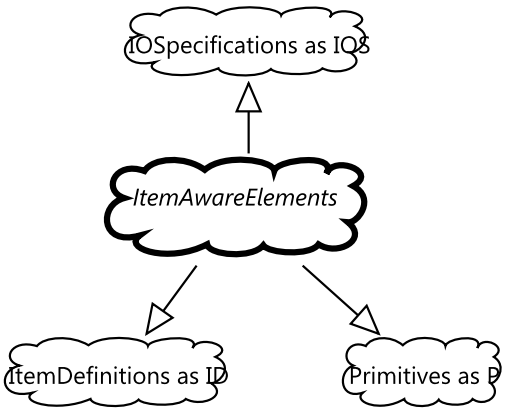


Figure B.46: The *`ItemAwareElements'`* package diagram

B.17 The IOSpecifications package

The invariants `InputSetReferenceIntegrity` and `OutputSetReferenceIntegrity` satisfy the constraints that, during execution, an `Activity` cannot refer to input, respectively output, that has not previously been included in the `dataInputRefs` and `dataOutputRefs` respectively. For simplicity reasons, only one of the symmetric requirements on input and output will be detailed. In VCL, this can be expressed using subsetting as both relations, `dataInputRefs` and `whileExecutingInputRefs`, form tuples of `InputSet` and `DataInput` elements. The same applies to restrictions on the definition of optional `DataInputs`.

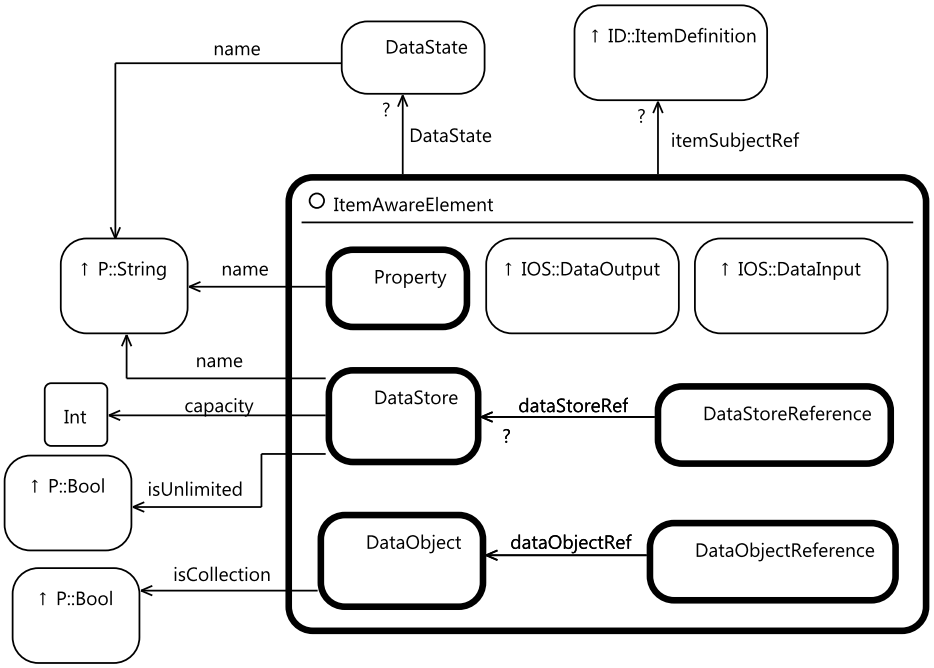


Figure B.47: The *ItemAwareElements'* structural diagram

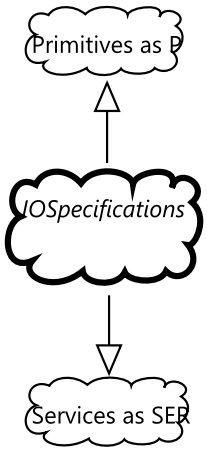


Figure B.48: The *IOSpecifications'* package diagram

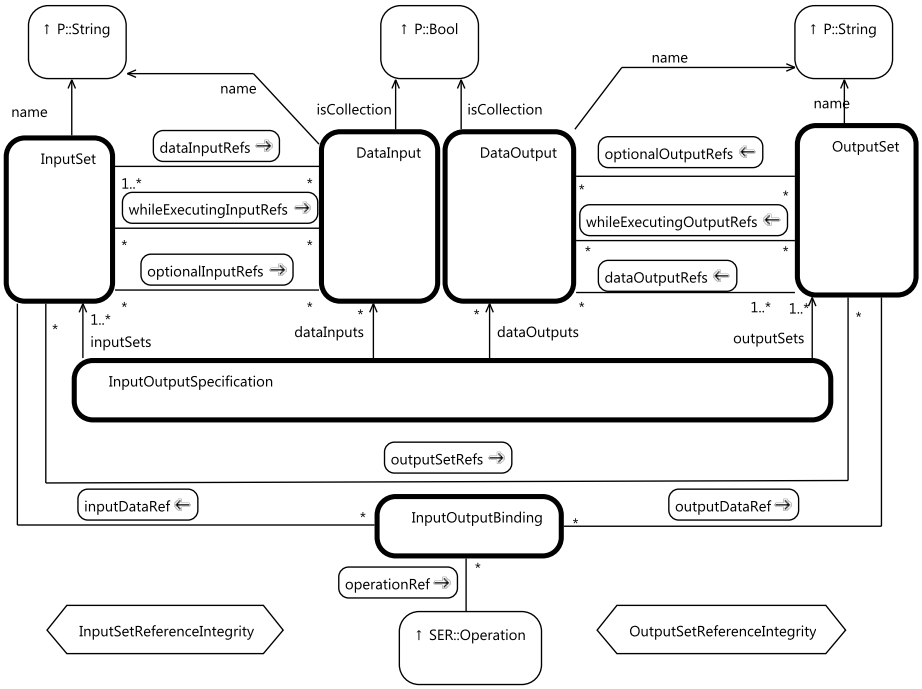


Figure B.49: The *IOSpecifications'* structural diagram

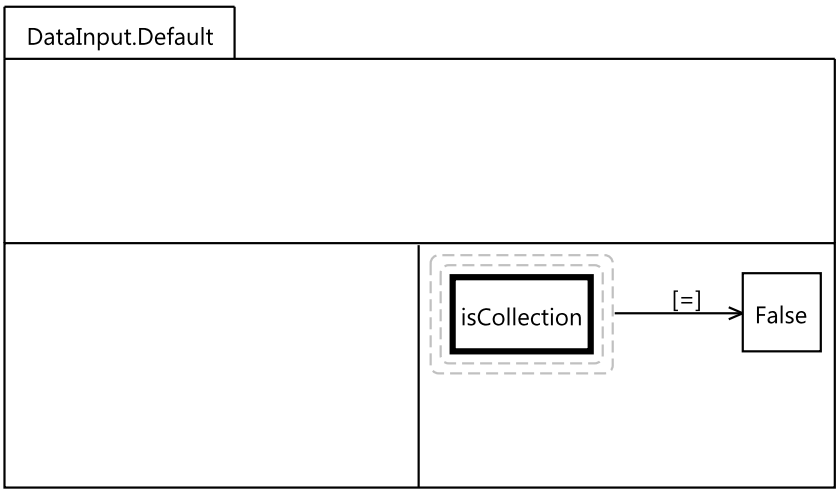


Figure B.50: The *Datainput's* `Default` operation

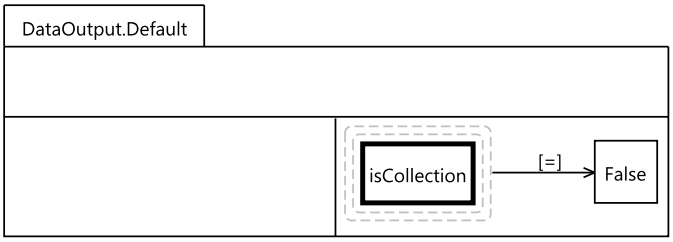


Figure B.51: The *DataOutput's* `Default` operation

Figure B.52: The `InputSetReferenceIntegrity` assertion expressing constraints on the `DataInput`

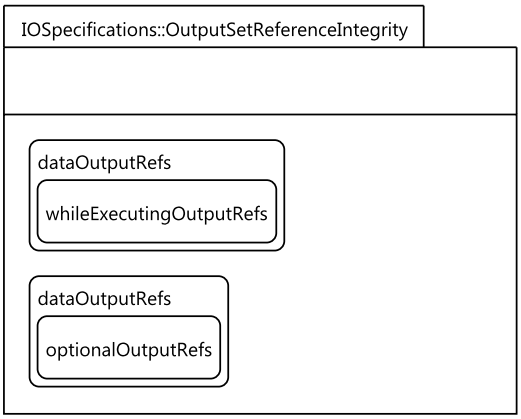


Figure B.53: The `OutputSetReferenceIntegrity` assertion expressing constraints on the `DataOutput`

B.18 The DataAssociations package

`DataInputAssociations` and `DataOutputAssociations`, respectively their union, being proper subsets of `DataAssociations` is expressed in VCL through insideness and the parent set being marked as defined through its subsets. A `transformation` may be specified which uses a `FormalExpression` to transform the data types. The specification puts constraints on the data types. The `ReferenceIntegrity` invariant addresses all these requirements by requiring that three nested invariants be satisfied. The `SingleSource` invariant expresses that, should there be no `transformation` defined for a `DataAssociation`, then it cannot have more than one source reference.

Furthermore, to assure the integrity of the data being referenced, either `DefinitionsMatch` or `AssignmentNotNull` must be satisfied. The first invariant expresses that definitions of the referenced source and target `ItemAwareElements` match. This is unfortunately not possible to express as VCL offers no quantifiers. Should types not match than it is required that, through the `AssignmentNotNull` invariant, the `assignment` not be null. An `Assignment` is a concept that uses an `Expression` to map data from one type to another. The specification does by no means specify that the `Expression` must actually perform an `Assignment` that would translate the data into an acceptable type. It is deemed the responsibility of the model defining the actual types and expressions to satisfy that constraint.

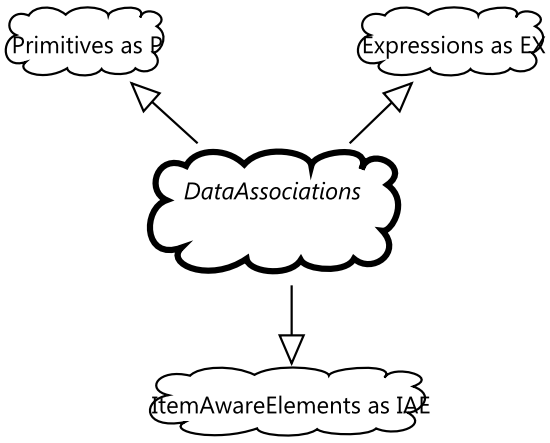


Figure B.54: The *DataAssociations'* package diagram

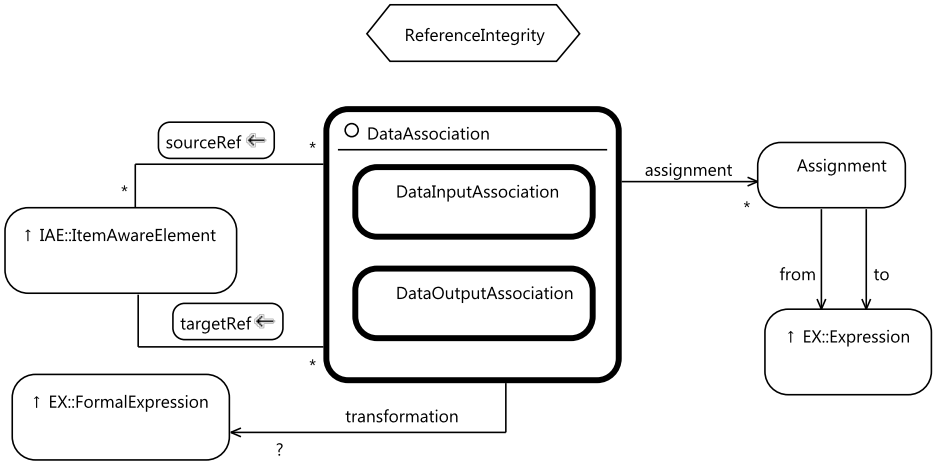


Figure B.55: The *DataAssociations'* structural diagram

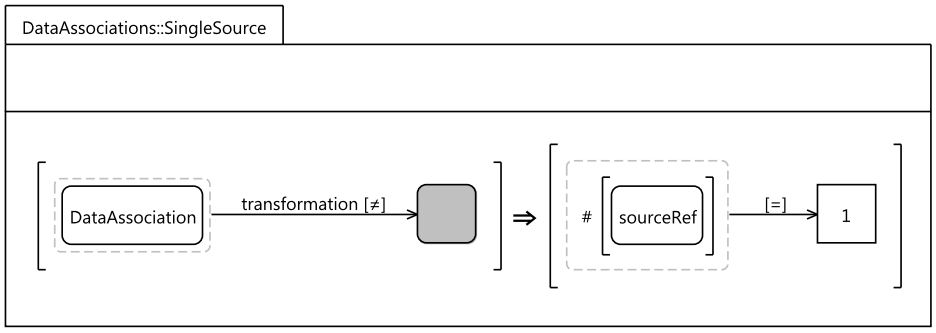


Figure B.56: The `SingleSource` assertion expressing a constraint on data sources

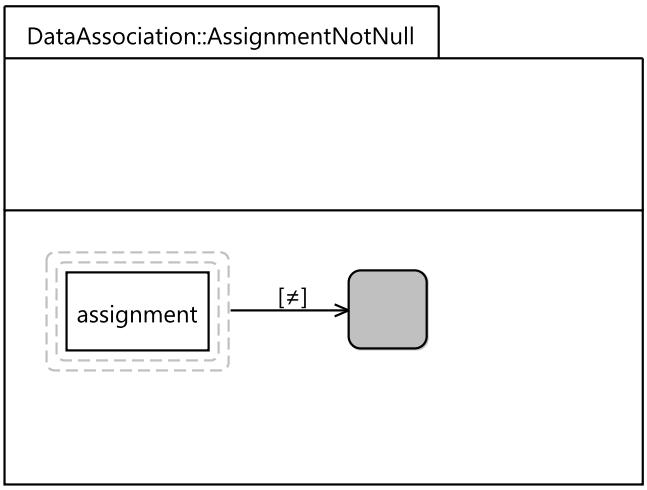


Figure B.57: The `AssignmentNotBull` assertion expressing that there must be an `Assignment`

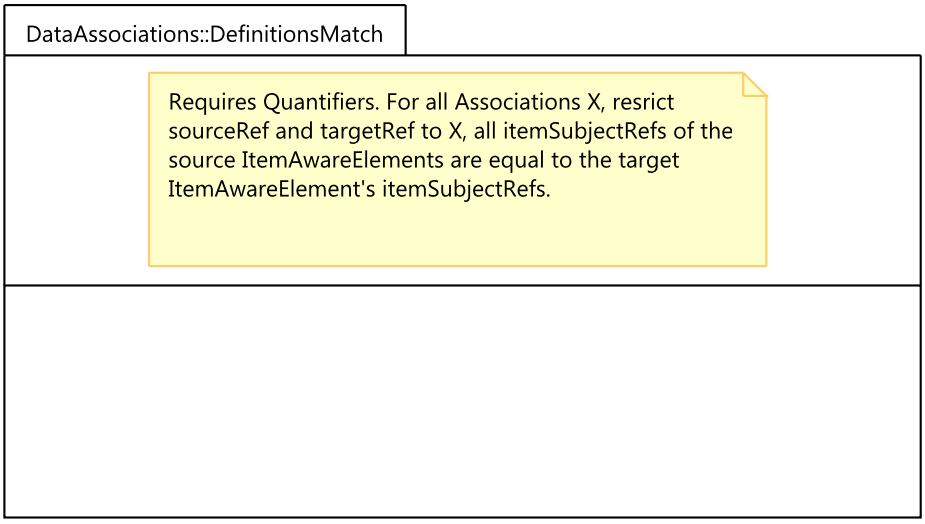


Figure B.58: The `DefinitionsMatch` assertion on matching `ItemDefinitions`

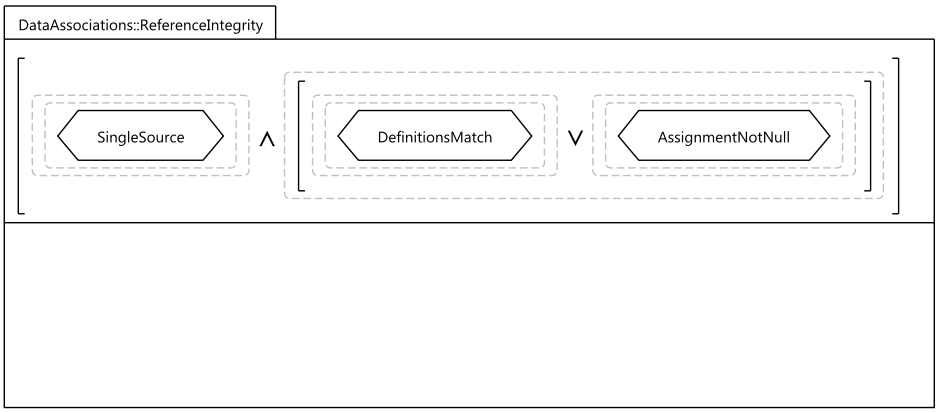


Figure B.59: The `ReferenceIntegrity` assertion expressing a constraint on data source and targets

B.19 The Data package

The *`Data`* VCL package groups all concerns from the *`ItemAwareElements`*, *`IOSpecifications`* and *`DataAssociations`* VCL packages. There is no corresponding BPMN2 package although a *`Data`* is hinted at on multiple occasions, especially in the chapter on items and data. The *`Data`* package expresses an invariant on the matching of the `isCollection` property of a `DataInput` and its referenced `ItemDefinition's isCollection` property.

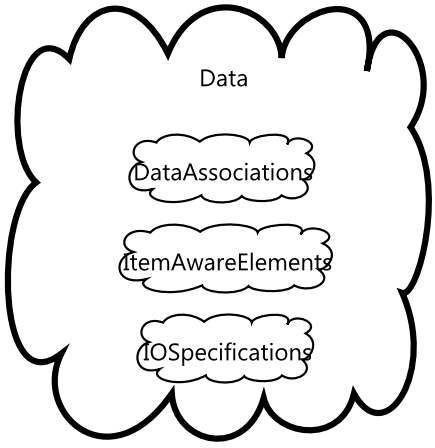


Figure B.60: The *`Data's`* package diagram

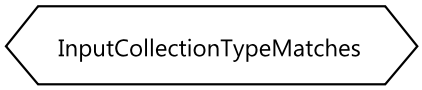


Figure B.61: The *Data's* structure diagram

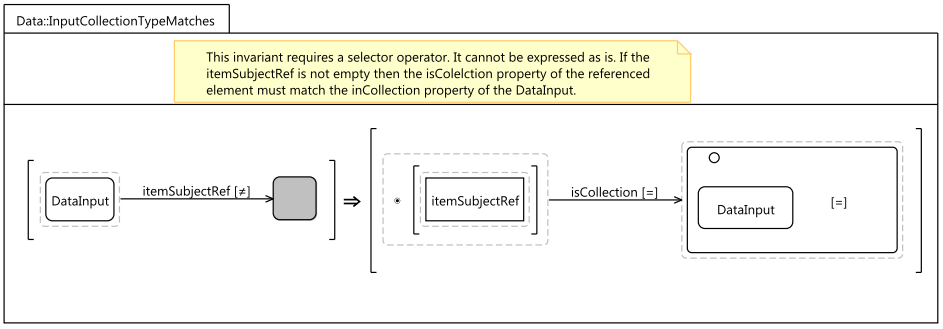


Figure B.62: The `Data's` assertion expressing constraints on the `DataInput`

B.20 The Processes package

Single `Processes` can be defined for a shepherding `Collaboration` when their flow is contained within a pool or `Lane`. The `Process'` visibility is given by its `ProcessType`. It may not be specified, `Public`, in which case it can be used within a `Collaboration`, exposing internal elements, or `Private` in which case it can only be used within an organisation's pool, not exposing any internals to the public.

`Processes` can be closed, not allowing any `Messages` to be received or send beyond the scope of the `Process`. This package also defines `Monitoring` and `Auditing` facilities which can be referenced by `Processes`. A `Process` may define any number of `Artifacts` that it contains and be bound to a `ResourceRole`, responsible for the `Process`. It can also define any number of `Properties`. The execution semantics for a `Process` are beyond the scope of this model. However, a constraint guaranteeing that, if a `Process` is executable, all `SequenceFlows` are executed atomically. This constraint is modelled by the `ExecutedImmediately` invariant expressed in the *`FlowElementContainers`* package in Section B.26.

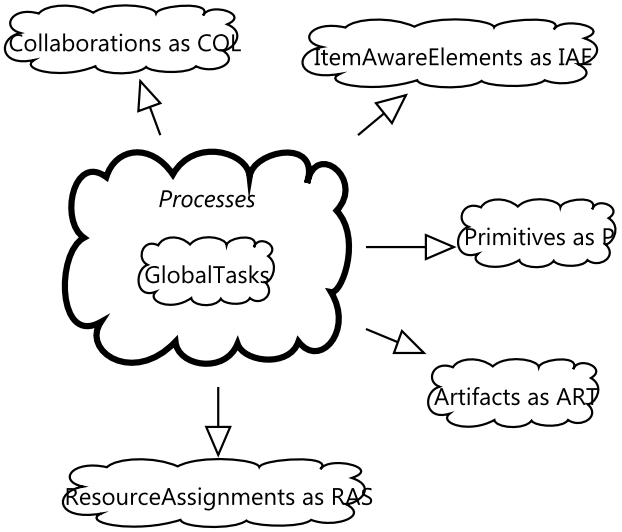


Figure B.63: The *Processes'* package diagram

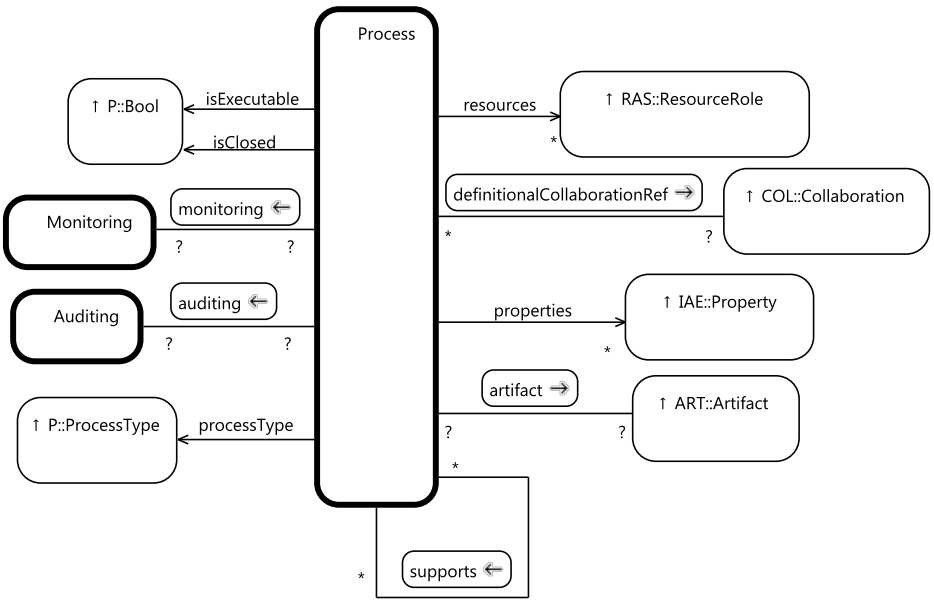


Figure B.64: The *Processes'* structure diagram

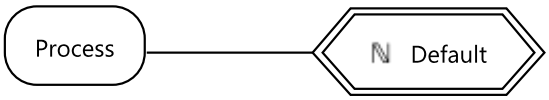


Figure B.65: The *Processes'* behaviour diagram

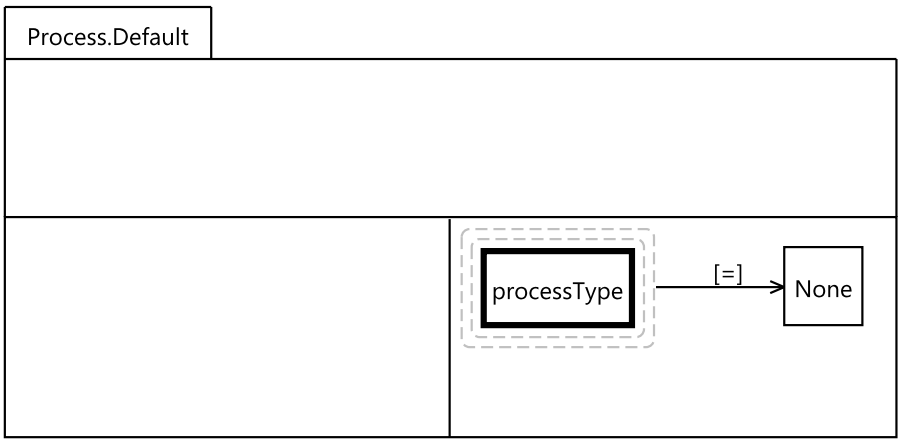


Figure B.66: The `Process' Default` operation

B.21 The Lanes package

`LaneSets` partition `Processes` to show roles. They can contain individual `Lanes`. `Lanes` in turn contain `FlowElements` and a reference to all `LaneSets` they contain. The `partitionElement` property and the corresponding reference property model a restriction mentioned by the specification. If added, `Lanes` can only contain `FlowElements` of the type of the `BaseElement` given as `partitionElement`. As this requirement would be modelled as an invariant in VCL and be in need for universal quantification, it has not been modelled.

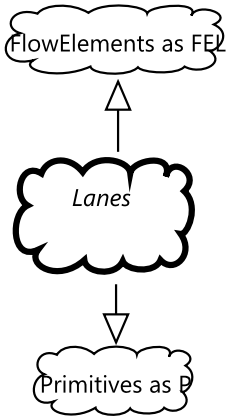


Figure B.67: The *Lanes'* package diagram

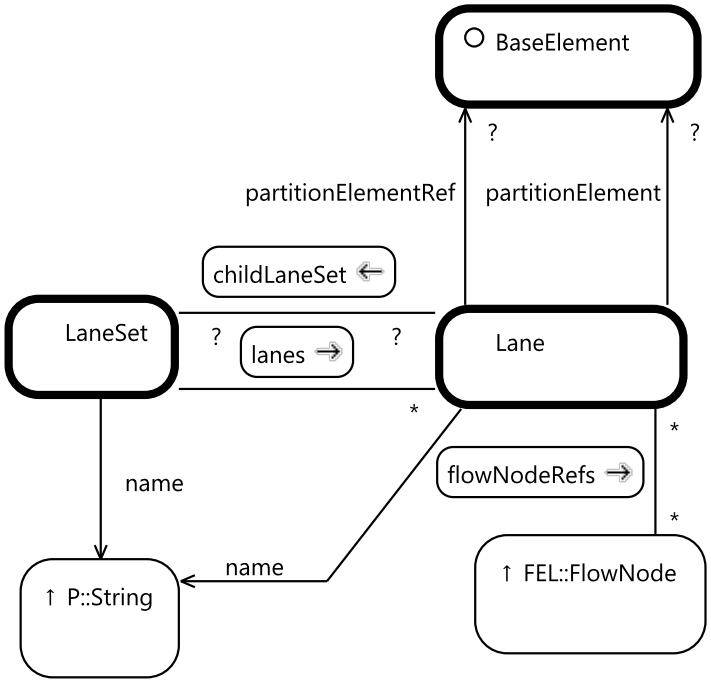


Figure B.68: The *Lanes'* structural diagram

B.22 The Escalations package

The *Escalations* VCL package is part of the *Event* VCL package as by the specification document and thereby is also a part of the *Common* VCL

package. Much like `Error`, `Escalation` packs an `ItemDefinitions`, a `name` and `escalationCode`. The constraints regarding the code as given by the specification document are expressed on three target `Events`. The specification mentions the result of the `Events` being a determining factor. Yet there is no concept modelling the result of an `Event`. This requirement will have to be modelled at a lower level unless the assumption is made that it should be modelled independent from the result which would be correct. However, it would impose a requirement that is not given by the specification.

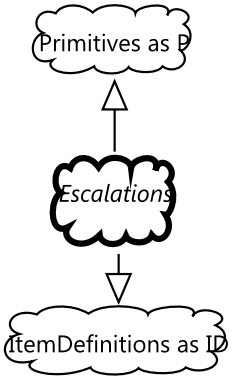


Figure B.69: The *Escalations'* package diagram

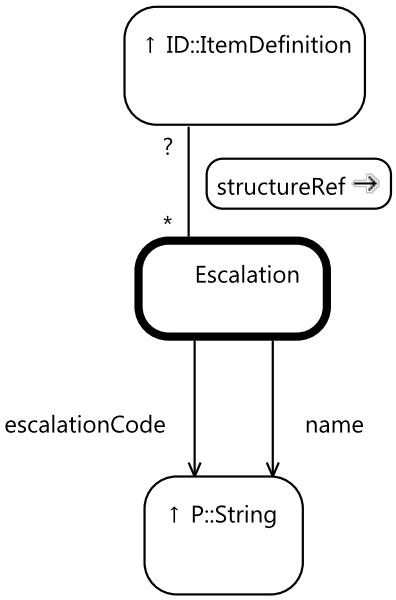


Figure B.70: The *Escalations'* structural diagram

B.23 The EventDefinitions package

The union of all subsets of `EventDefinitions` forms a proper subset, expressed through the ◯ symbol on the `EventDefinitions` blob. `ErrorEventDefinition`, `EscalationEventDefinition`, `MessageEventDefinition` and `SignalEventDefinition` each pack a reference to the object they are carrying. A `CompensateEventDefinition` designates an `Activity` that is used to cancel respectively undo changes induced by the throwing `Process` or `SubProcess`. `ConditionalEventDefinitions` specify an `Expression` that must evaluate to `True` in order for the `Event` to take place. `CancellationEventDefinitions` are used on `Transaction SubProcesses` usually in conjunction with a `CompensationEventDefinition` to stop its execution and handle the cancellation.

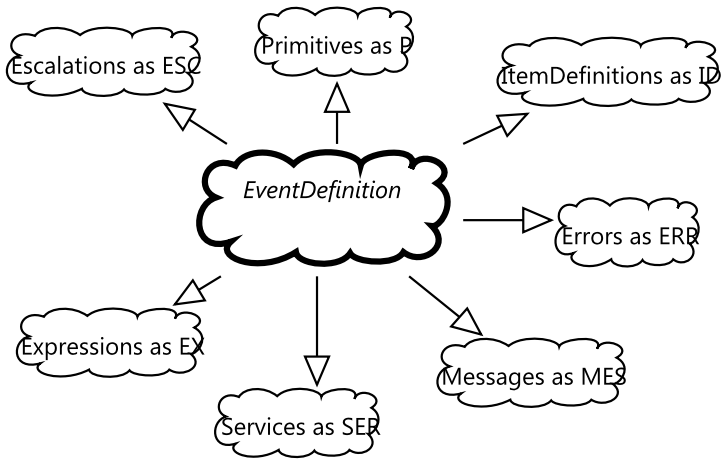


Figure B.71: The *EventDefinitions'* package diagram

`LinkEventDefinitions` are always used in pairs and are used to visually simplify the model by logically linking, using a name, two `Events` on the same `Process` level. They effectively function as "Go To `Events`. `SignalEventDefinitions` reference a `Signal`, also defined in this package. The `Signal` carries a payload as defined by an `ItemDefinition`. A `TimerEventDefinition` specifies, through multiple `Expressions`, a date, duration and cycle, allowing for timed `Events` to be handled. As the specification lacks detail, `TerminateEventDefinition` has not been modelled. Moreover, due to the lack of quantifiers and related problems realising constraints, instead of implementing the requirements only partially, no invariants were modelled for this package. The specification defines two special types of `Events` that have either no `EventDefinition` or multiple ones. While these are con-

cepts that have a visual representation, they are implicit by the reference of `EventDefinitions` to the `Event`. As such, they are not modelled as they only have a semantic meaning.

The BD exposes one global operation which is used be the *`SubProcess'`* `OneStartEvent` invariant to retrieve the subset of all `EventDefinitions` that are allowed to figure on its `StartEvent`. The `GetValidSubProcessStarts` operation expressed the union of all allowed `EventDefinitions` and returns them as the `ValidStarts` set.

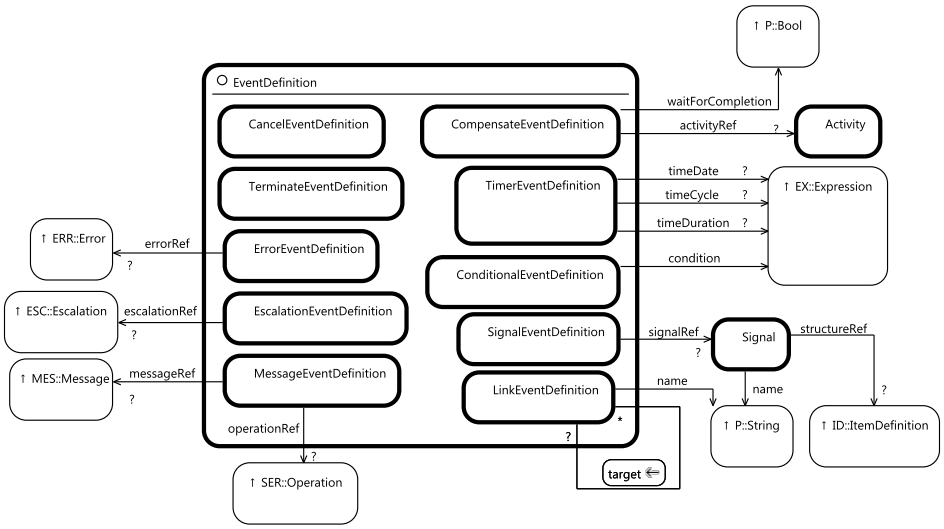


Figure B.72: The *`EventDefinitions'`* structural diagram

Figure B.73: The *`EventDefinitions'`* behavioural diagram

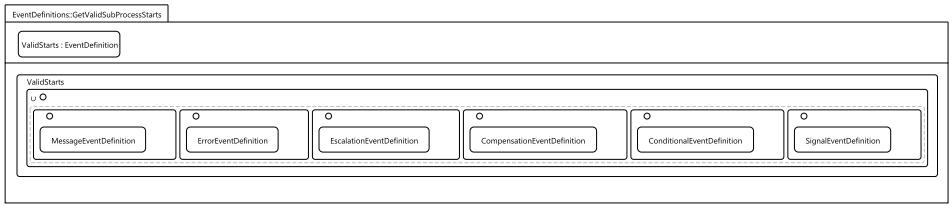


Figure B.74: The `GetValidSubProcessStarts` assertion

B.24 The Events package

`ThrowEvents` and `CatchEvents` are subsets of `Event` as expressed by the insideness on the VCL structure diagram. Each `Event` can have several `Properties`. Every `Event` defines a reference to an `EventDefinition`. This is what defines the kind of activity the `Event`will produce or react to. While the reference holds the top level `EventDefinition`, the `Event` also holds a reference to the particular `EventDefinition` that will trigger it. By this scheme, triggering or halting events can be exchanged for other, identically typed, events without too much of a hassle. Each `Event` is also tied to either a `DataInputAssociation` respectively a `DataOutputAssociation` and either `DataInput` or `DataOutput` depending on the concrete `Event` subset. The data associations are used to assign `Data` from the `Event` to a `Data` element and vice versa. Moreover, the `Event` also hold references to the actual sets of data objects.

The `StartEvent` is the entry point of a `Process` and is a specialised `CatchEvent`. While the specification is much more extensive on explaining the different `Events` as well as introducing some additional semantic elements such as tokens, this document will not consider those concepts as they are semantic constructs and not modelled and considered beyond the scope of this document. The Boolean property of the `StartEvent` is used in some types of `SubProcesseses` to restart the `SubProcess` every time its `StartEvent` is triggered or allow for multiple, parallel instances to run. The `EndEvent` serves as an end point to `Processes`, halting all `SequenceFlows`. `StartEvents` have a trigger and `EndEvents` a result as given by the referenced `EventDefinition`.

`IntermediateCatchEvents` and `IntermediateThrowEvents` can, as their name suggests, happen as the `Process` flow is executed. `IntermediateThrowEvents` serves to produce `Events` and resume normal process flow. The type of `Event` that is produced depends on the referenced `EventDefinition`. `IntermediateCatchEvents` halt the process flow and wait for the trigger,

as given by the referenced `EventDefinition` to be caught before allowing the process flow to resume. `ImplicitThrowEvent` are a specialised form of `ThrowEvents` in that they have no visual representation. They are mainly used on multi-instance `Activities` to not introduce additional graphical overhead.

`BoundaryEvents` have a boolean property `cancelActivity` which indicates whether the `BoundaryEvent` will cause the `Activity` to halt when it is triggered. This property is subject to an invariant, `Interrupting-Behaviour`, specifying that all `BoundaryEvenets` that cancel the `Activity` they are attached to must reference an `ErrorEventDefinition`. Another global invariant, `ConsistencyRequirement`, states that `Processes` without a `StartEvent` cannot have an `EndEvent`. The requirements on `Data` transfer and behaviour during `Events` have not been implemented.

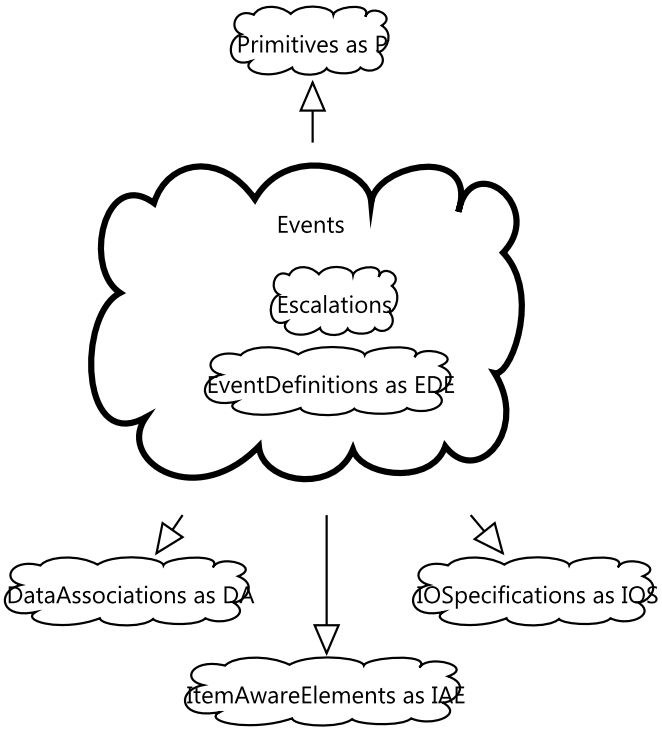


Figure B.75: The *Events'* package diagram

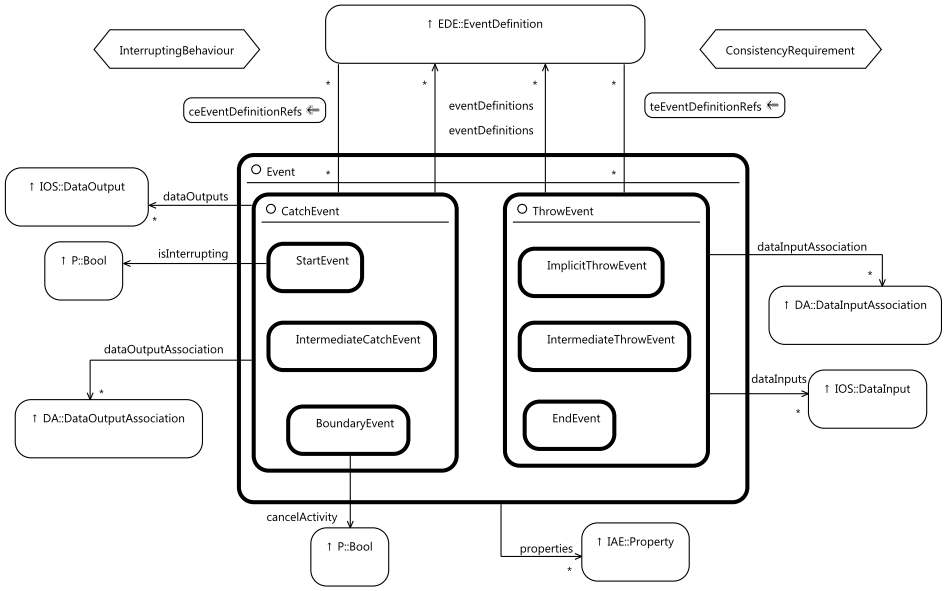


Figure B.76: The *Events'* structural diagram

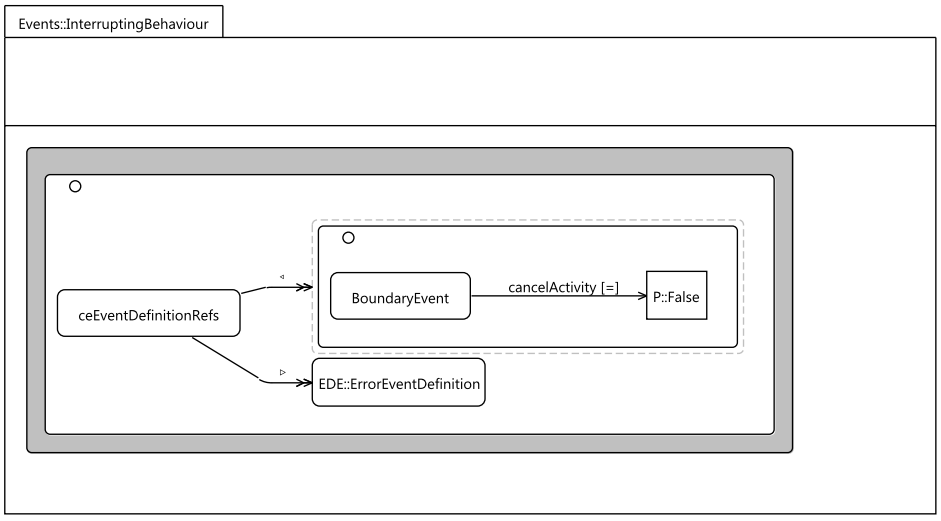


Figure B.77: The `InterruptingBahaviour` assertion expressing constraints on `BoundaryEvents`

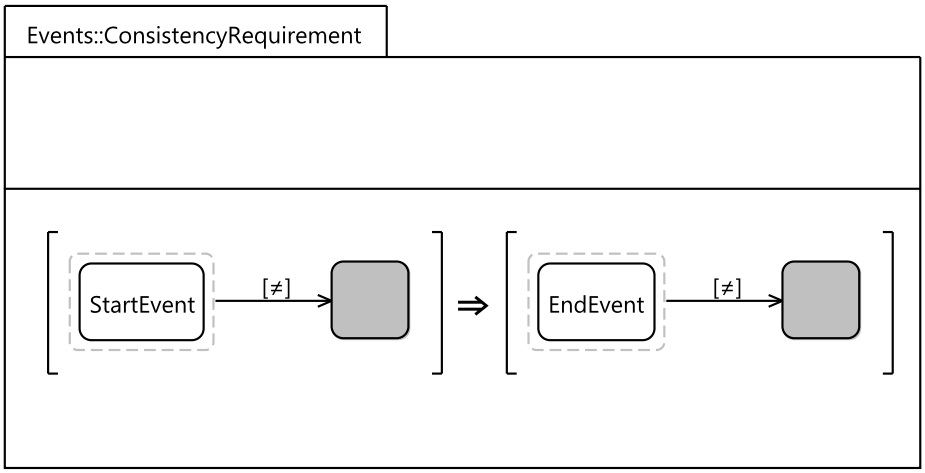


Figure B.78: The `ConsistencyRequirement` assertion

B.25 The FlowElements package

The *`FlowElements`* VCL package groups all concepts from business process flows. All of these concepts can appear in `Process` flows. `FlowElements` is the overarching concept, encompassing more concrete concepts such as `FlowNodes`, `SequenceFlows`, `DataObjects` and `DataStoreReferences`. The specification document is ambiguous when it comes to `DataStoreReference` being a `FlowElement`. While it is displayed on the `FlowElement's` class diagram on page 87, the introduction to the section mentions `DataAssociations`, not `DataStoreReference`. It was chosen to model the latter. All `FlowElements` have a `name`, a reference to the `CategoryValue` that the `FlowElements` has been assigned as well as references to `Auditing` and `Monitoring` defined by the containing `Processes`.

`SequenceFlows` are used to instil an order in the arrangement of other `FlowElements`. They map a source `FlowNode` to a target `FlowNode`. A `SequenceFlow` may specify a conditional `Expressions` who has to evaluate to `True` in order for the `SequenceFlow` to be a valid path. They are common on `SequenceFlows` hailing from `Gateways` to represent branching business decisions. The `isImmediate` property indicates whether the `SequenceFlow` is atomic, that is, if any `Activities` are allowed to occur between its source and its target. Through insideness, `Activities`, `Gateways` and `Events` are `FlowNodes`. The relations `incoming` and `outgoing` represent the inverse relations to `incoming` and `outgoing`.

The *FlowElement* package contains several invariants as well as a query operation. The GetCollaborationGivenFlowNode is used for what its name suggests. It is needed in the MessageFlowsSpanPools invariant. The EventSubProcessUnconnected invariant satisfies the requirement that SubProcesses triggered by Events cannot be the target or source of SequenceFlows. The SeuqenceFlowAttributeConstraints invariant satisfies the constraint that a StartEvent cannot have an outgoing SequenceFlow that is laden with a conditional Expression.

The SequenceFlowSourceConstraints and SequenceFlowTargetConstraints invariants each have three nested invariants, modelling the requirements on source and target Events for SequenceFlows. Due to the redundancy of many requirements on IntermediateEvents, their respective invariants have only partially be modelled. The invariants satisfy the requirement that no StartEvent may be the target of a SequenceFlow and that no EndEvent can be the source of a SequenceFlow. Furthermore, no BoundaryEvent can be the target of a SequenceFlow. The invariants also satisfy that a BoundaryEvent being used as compensation must have an outgoing SequenceFlow. And lastly, the invariants satisfy the requirement for IntermediateThrowEvents and IntermediateCatchEvents not to be the end or start of a flow, in other words, to have both, incoming and outgoing SequenceFlows. As there are many intermediate events, not all constraints have been implemented. Those modelled serve as a proof of concept and the remaining constraints have been omitted due to the redundancy of the modelling task and in the interest of conserving space and time.

The MessageFlowsSpanPools invariant unfortunately fails to satisfy the requirement that MessageFlows must connect two separate pools respectively InteractionNodes contained therein. This is due to the lack of quantifiers in VCL. In the absence of quantifiers, MessageFlowsSpanPools specifies a pair of different FlowNodes and Participants as input as well as a single MessageFlow object. For the invariant to work, these would need to be specified using existential quantifiers. The assertion has been modelled in order to be able to better understand and judge the shortcoming of VCL. These input objects are used in a chain of assertions to retrieve the Collaborations they would be contained in. The predicate compartment of the invariant then states that should those input FlowNodes and Participants be referred in an exchanged of Messages, that they need to be contained in different Collaborations which model the concept of pools, satisfying the requirement.

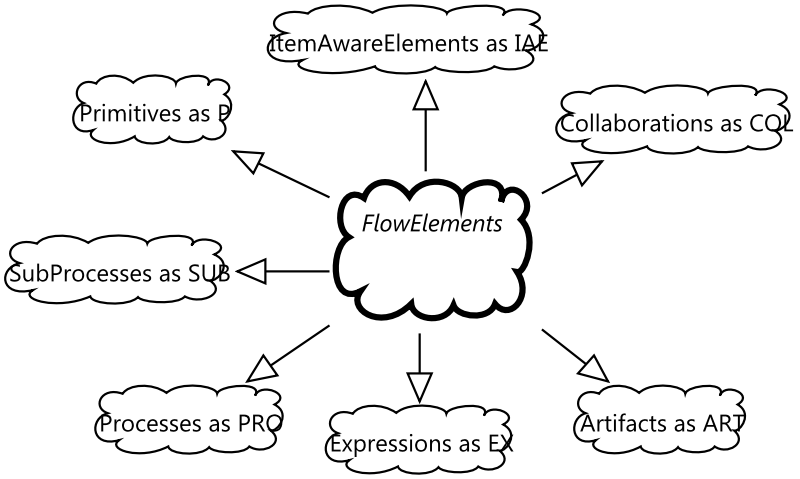


Figure B.79: The *FlowElements'* package diagram

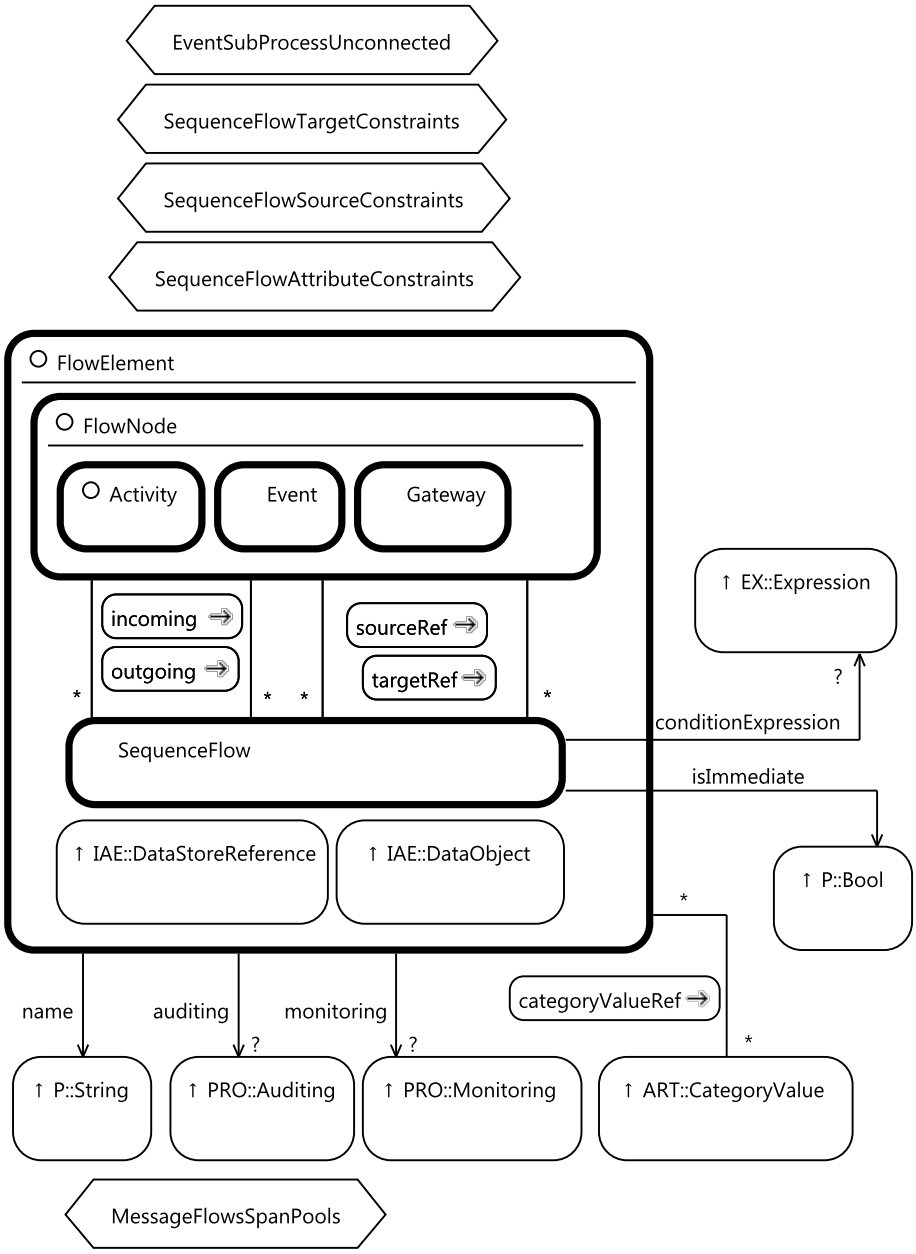


Figure B.80: The *FlowElements'* structural diagram

Figure B.81: The *FlowElements'* behaviour diagram

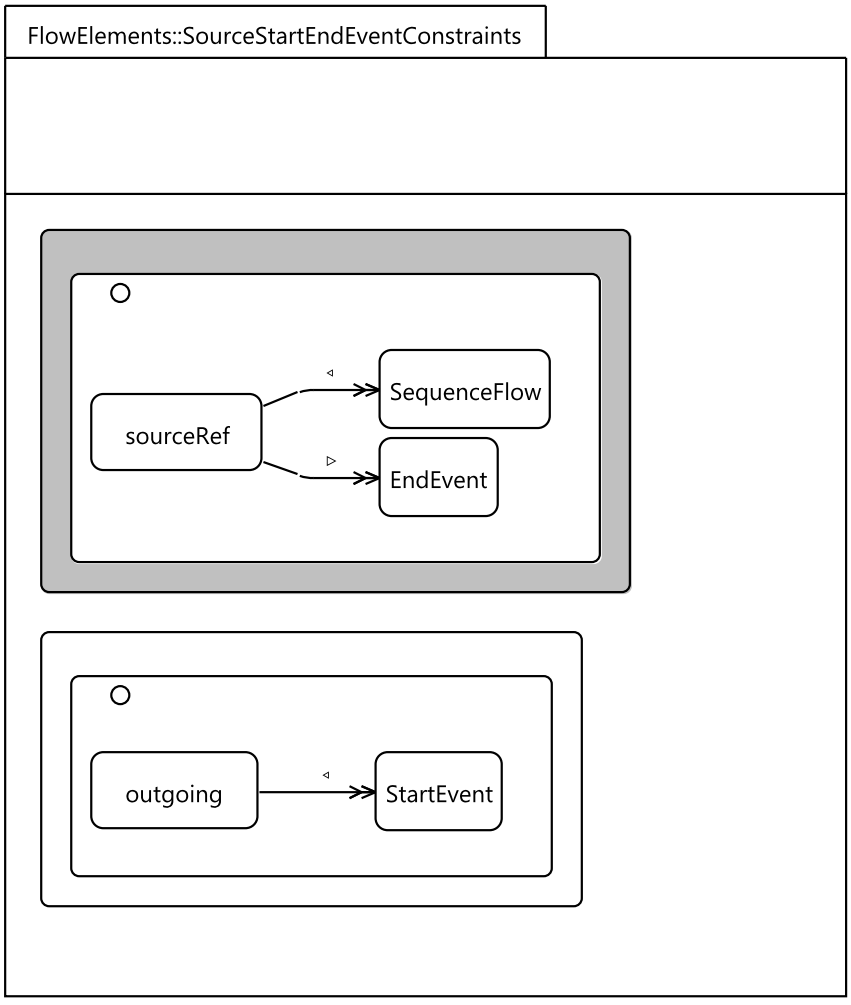


Figure B.82: The `SourceStartEndEventConstraints` assertion expressing constraints on sequence flow sources

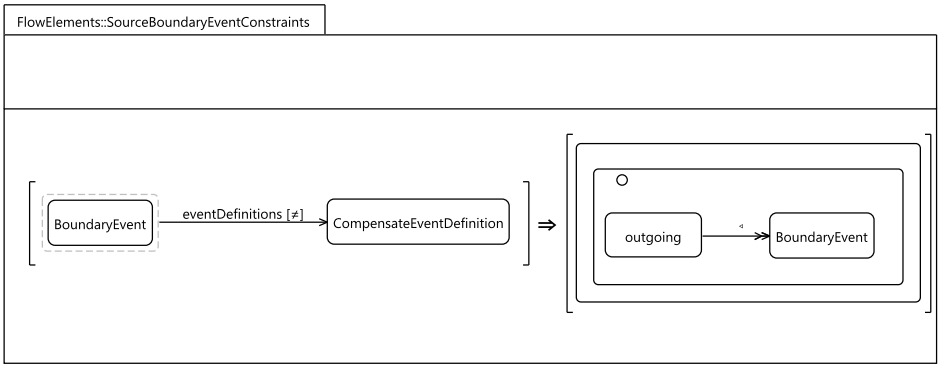


Figure B.83: The `SourceBoundaryEventConstraints` assertion expressing constraints on sequence flow sources

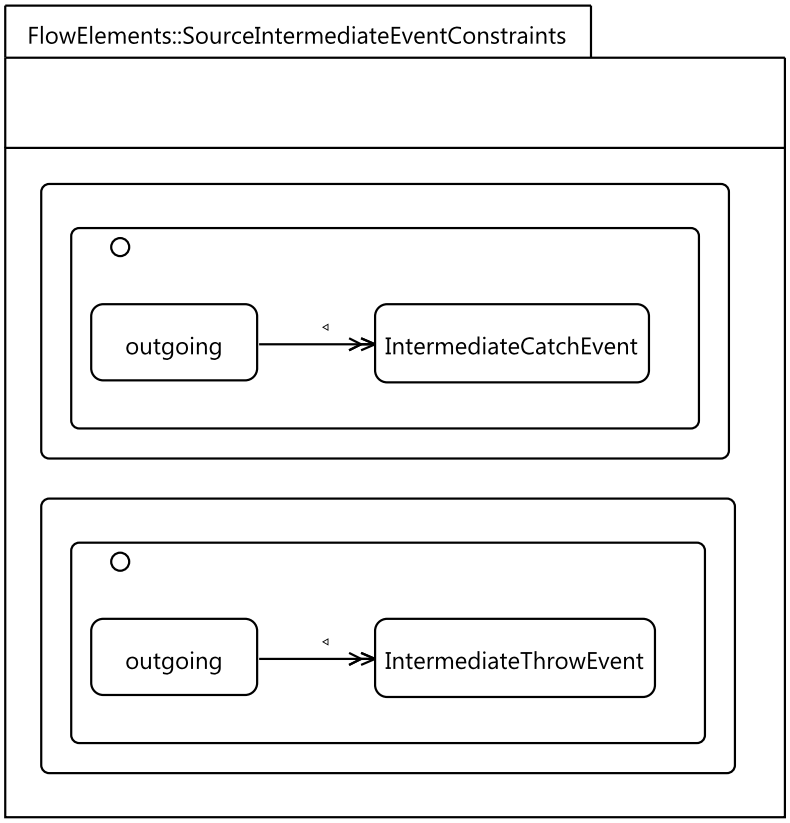


Figure B.84: The `SourceIntermediateEventConstraints` assertion expressing constraints on sequence flow sources

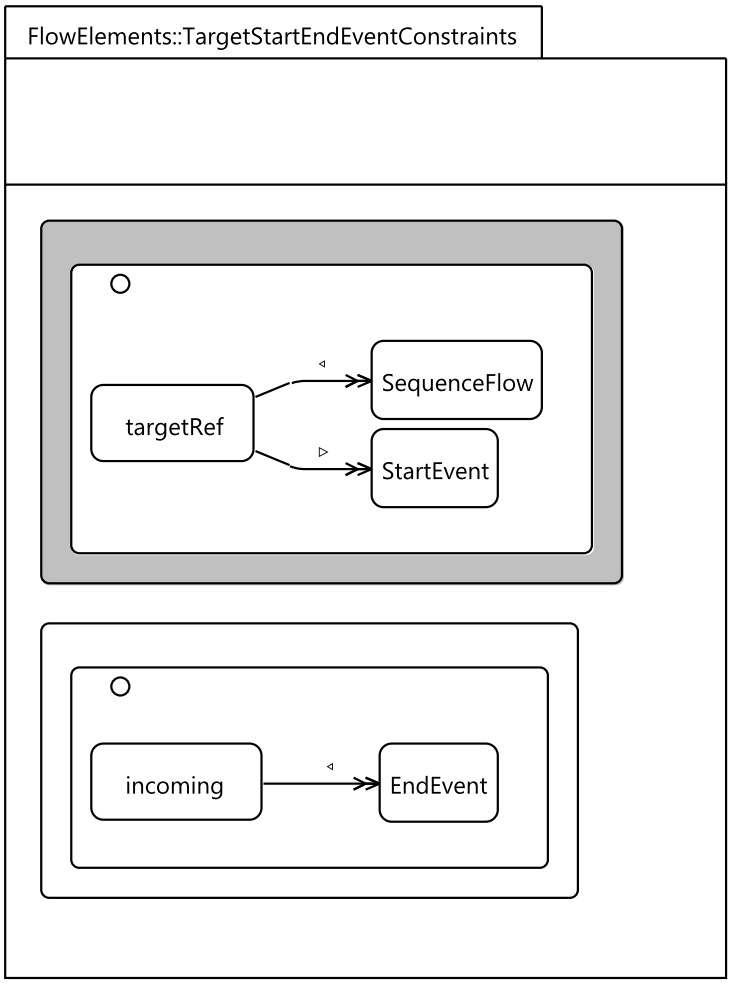


Figure B.85: The `TargetStartEndEventConstraints` assertion expressing constraints on sequence flow targets

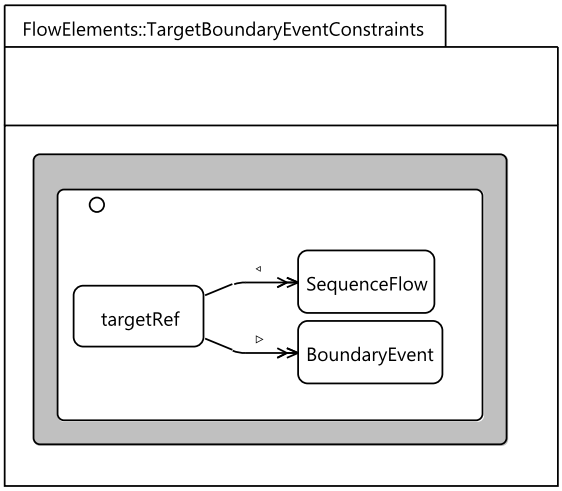


Figure B.86: The `TargetBoundaryEventConstraints` assertion expressing constraints on sequence flow targets

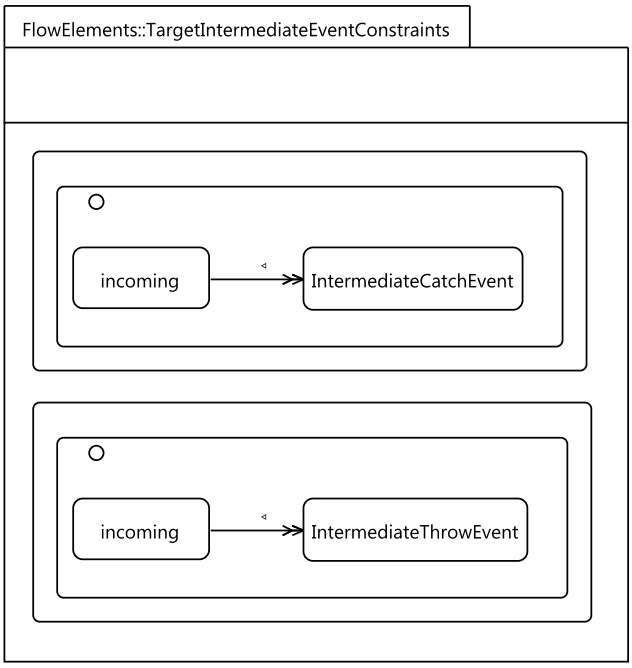


Figure B.87: The `TargetIntermediateEventConstraints` assertion expressing constraints on sequence flow targets

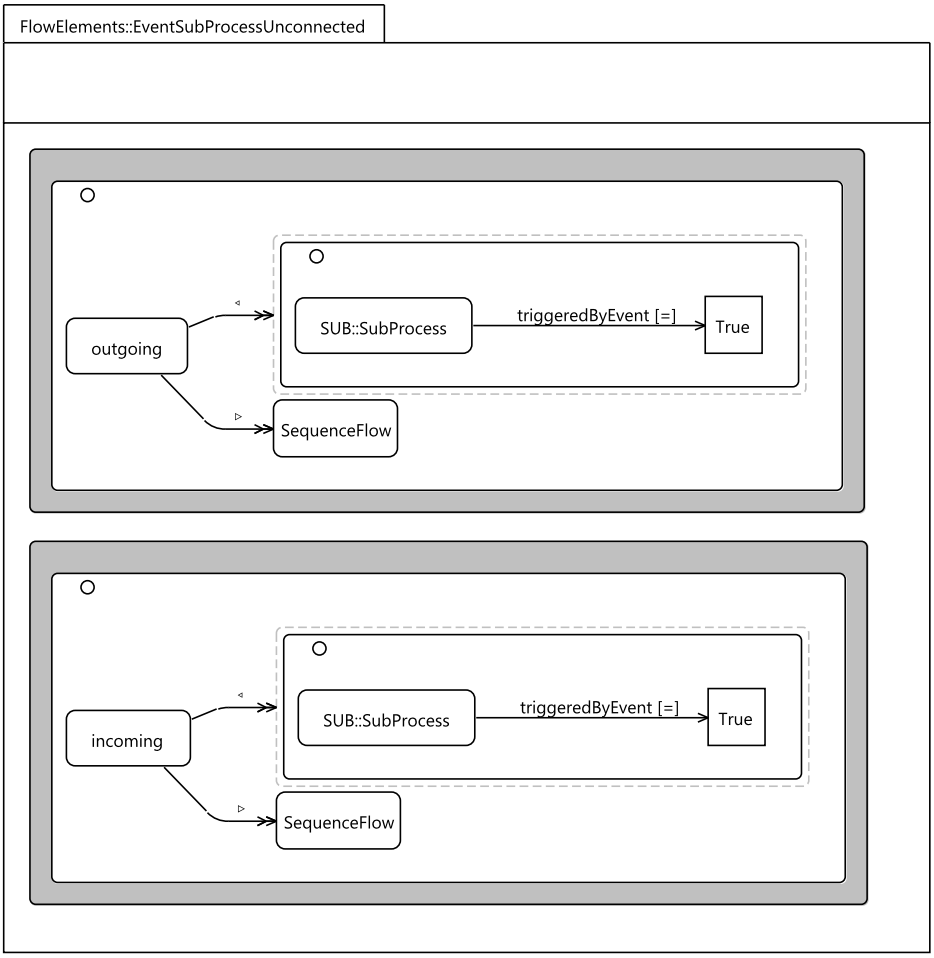


Figure B.88: The `EventSubProcessUnconnected` assertion expressing constraints on `SubProcess` connectivity

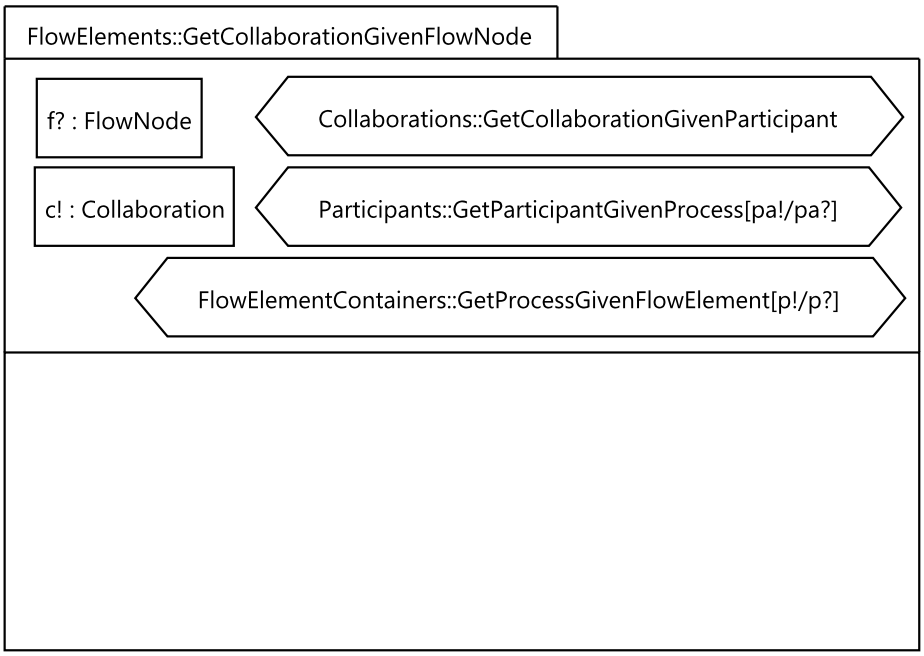


Figure B.89: The *FlowElements'* `GetCollaborationGivenFlowNode` operation

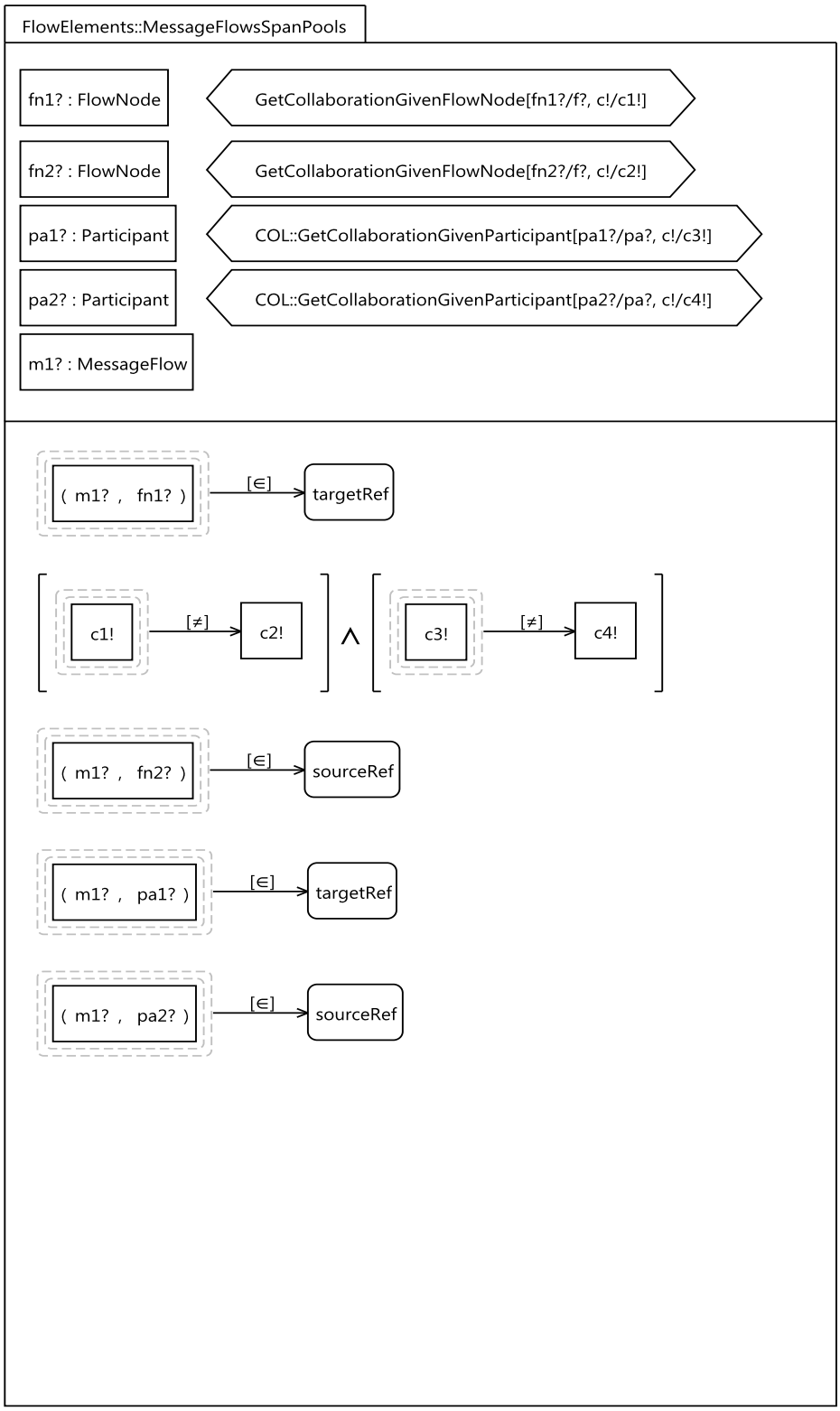


Figure B.90: The `MessageFlowsSpanPools` assertion expressing constraints on the `MessageFlow`

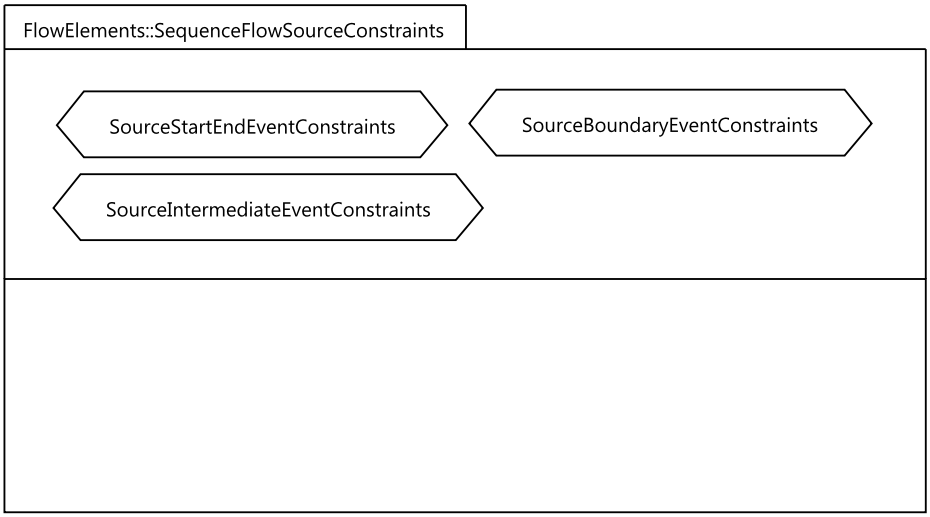


Figure B.91: The `SequenceFlowSourceConstraints` assertion expressing constraints on sequence flow sources

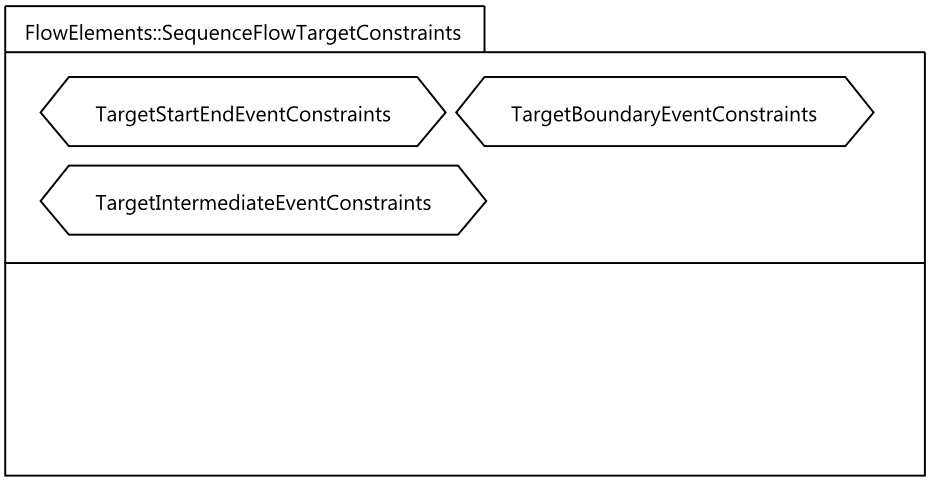


Figure B.92: The `SequenceFlowTargetConstaints` assertion expressing constraints on sequence flow targets

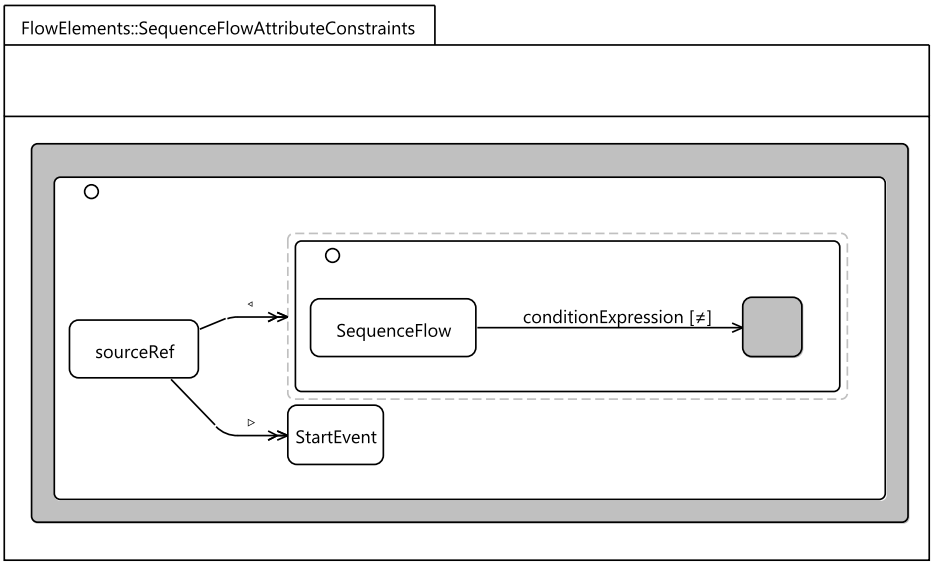


Figure B.93: The `SequenceFlowAttributeConstraints` assertion expressing constraints on sequence flow attributes

B.26 The FlowElementContainers package

The *`FlowElementContainers`* VCL package introduces a superset of container elements on BPMN2 diagrams. `FlowElementsContainer` is defined though two subsets, `Process` and `SubProcess`. These elements, by the `flowElements` relational edge, contain `FlowElements`. Moreover, Lanes can also be contained in `FlowElementsContainers`. A constraint is modelled by the `ExecutedImmediately` invariant expresses that if a `Process` is executable, all `SequenceFlows` are executed atomically. The global `GetProcessGivenFlowElement` operation returns the `Process` given an input `FlowElement`.

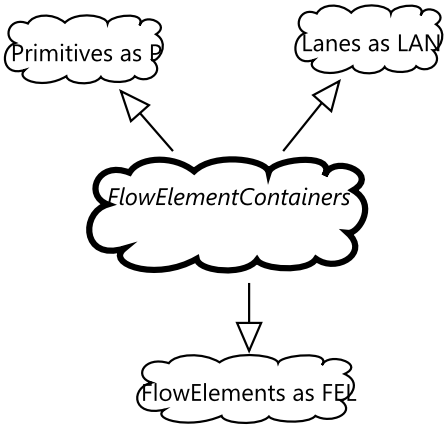


Figure B.94: The *FlowElementContainers'* package diagram

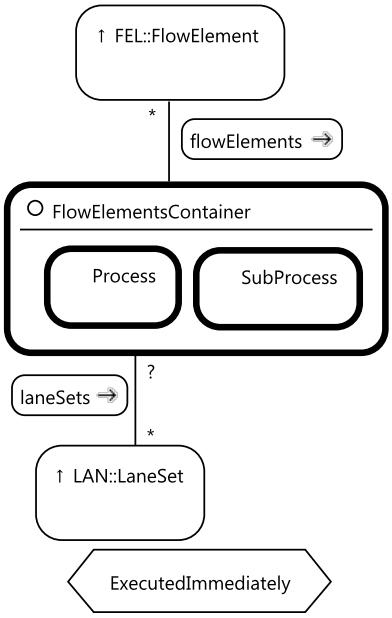


Figure B.95: The *FlowElementContainers'* structural diagram

Figure B.96: The *FlowElementContainers'* behavioural diagram

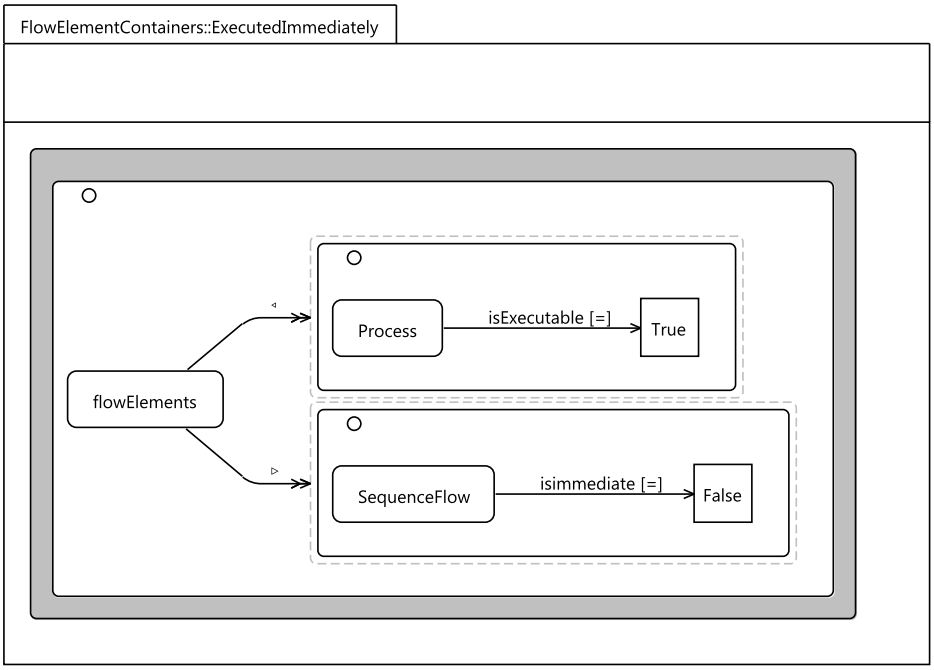


Figure B.97: The `ExecutedImmediately` invariant expressing constraints on `Processes` execution

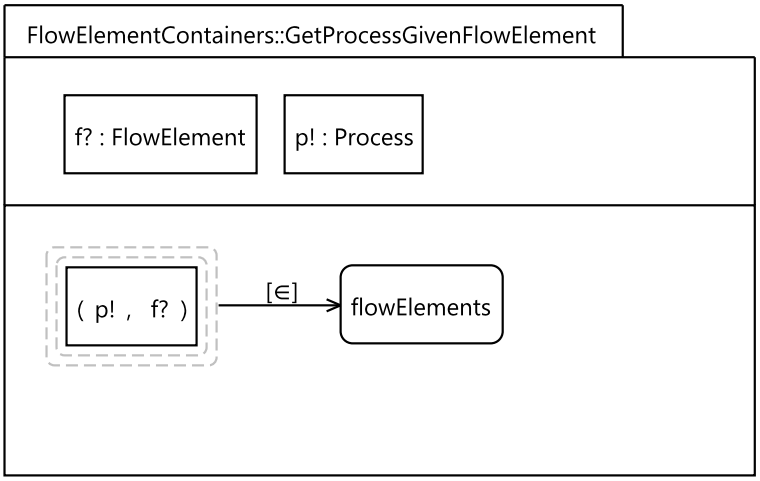


Figure B.98: The `Process' GetProcessGivenFlowElement` operation

B.27 The CallableElements package

`CallableElement` is defined through its two subsets, `Process` and `GlobalTask`. Referenced `Interfaces` can define the `Operations` defined by the named `CallableElement`. Binding of input respectively output is defined through an `InputOutputSpecification`. An `InputOutputBinding` binds inputs and outputs for use in `Operations`.

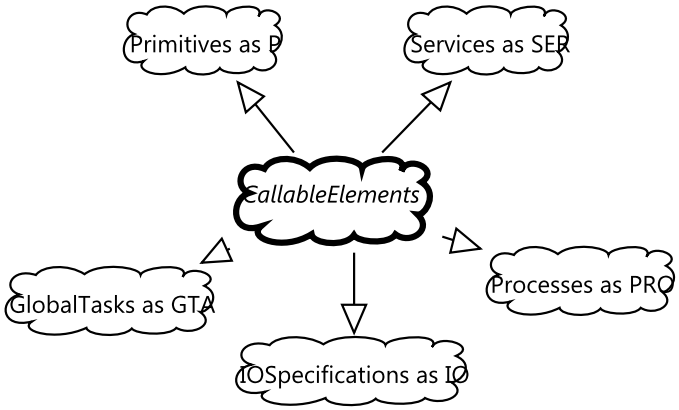


Figure B.99: The *CallableElements''* package diagram

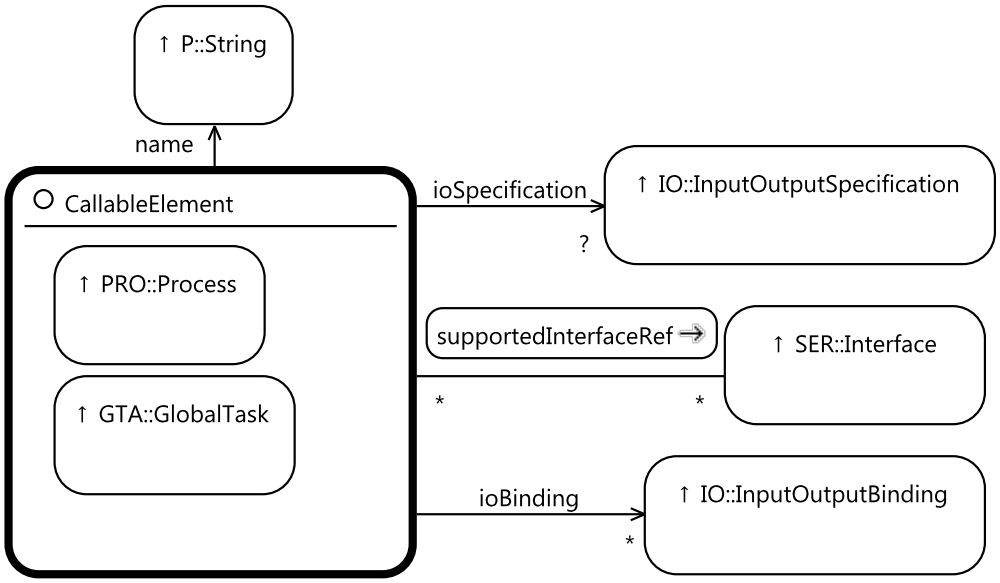


Figure B.100: The *CallableElements''* structural diagram

B.28 The Common package

The *Common* VCL ensemble package logically group other packages. *Common* includes all packages that specify concepts used across any diagram types in BPMN2. It groups all foreign packages contained in its package.

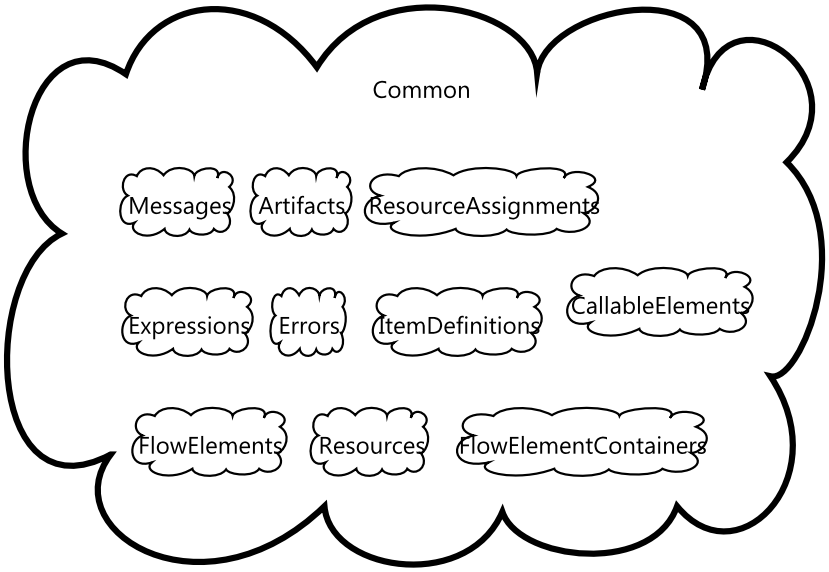


Figure B.101: The *Common's* package diagram

B.29 The LoopCharacteristics package

`LoopCharacteristics` are defined by two concrete subsets, `StandardLoopCharacteristic` and `MILCharacteristic`. The first introduces the standard loop concept, looping on a boolean condition, usually a counter. The metamodel does not contain runtime attributes as mentioned by the specification. The `MILCharacteristic` models complexer looping behaviour. "MIL" is short for "multi-instance loop". `Activities` defining a `MILCharacteristic` can run in parallel, not requiring a sequential execution of loops. The loops are engaged either by an `Expression` or by a provided `ItemAwareElement`. Referenced `DataInputs` and `DataOutputs` are used to provide individual hooks for each instance to receive and produce `Data`.

`Events` occurring or received during an `Activity` currently executing in a multi-instance loop display complex behaviour. This is handled by the `MultiInstanceBehaviour` property which specify that `None`, `One`, `All`, or any number of `Events` are handled during the loop. The latter requires the consultation of a `ComplexBehaviourDefinition`.

The `MILCharacteristic's Instantiation` invariant satisfies the constraint that if a `MILCharacteristic` specifies a `loopCardinality`, then it must not specify an `ItemAwareElement` that is to regulate the number of loops and vice versa. The constraint is loosely interpreted as the specification mentions that "either-or" which has been modelled as a local invariant, `Instantiation`, stating that either the `loopCardinality` refers to an empty set or `loopDataInputRef`, but not both.

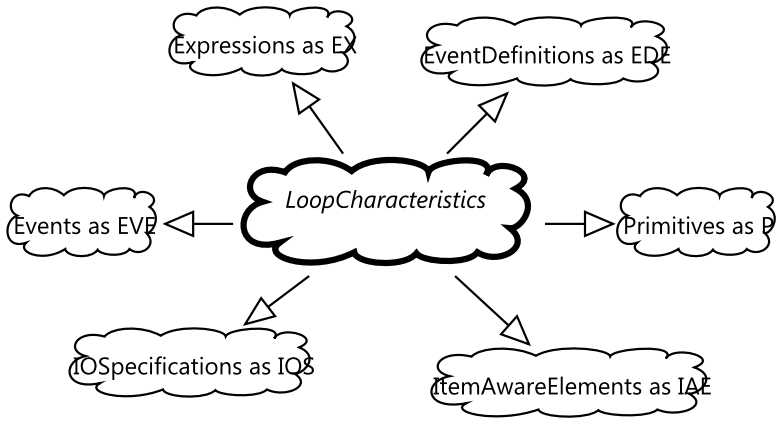


Figure B.102: The *LoopCharacteristics'* package diagram

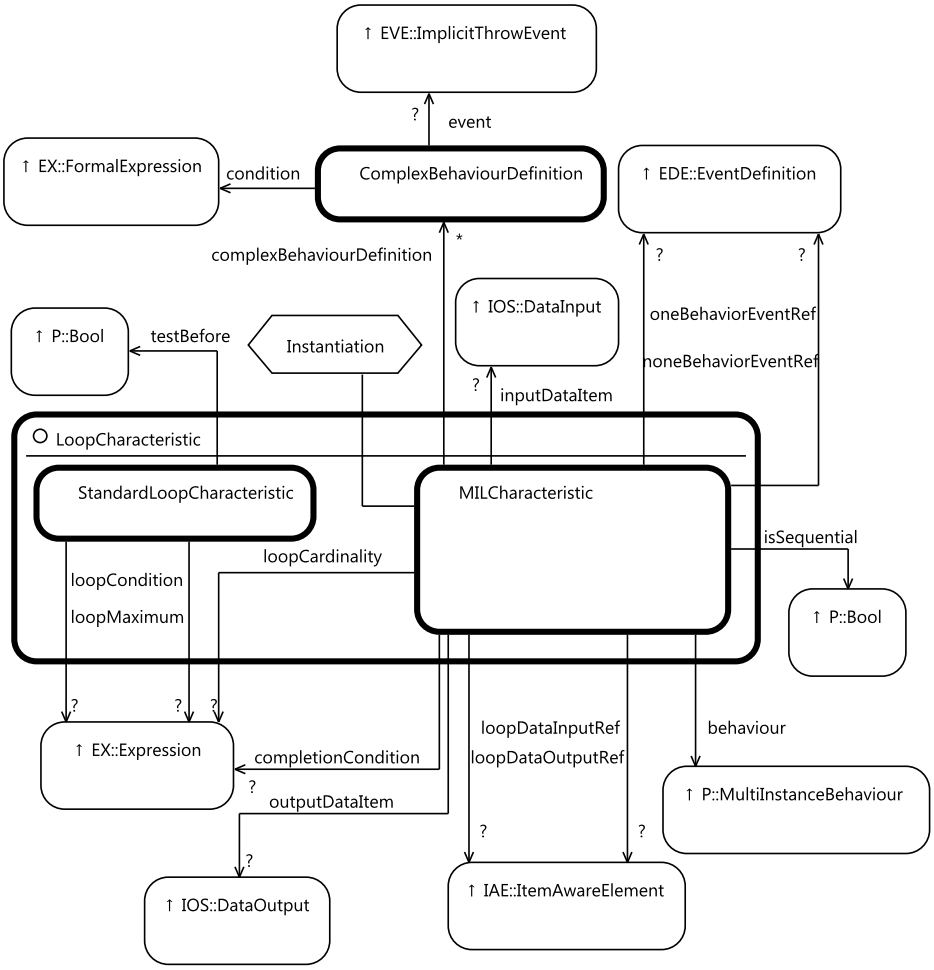


Figure B.103: The *LoopCharacteristics'* structural diagram

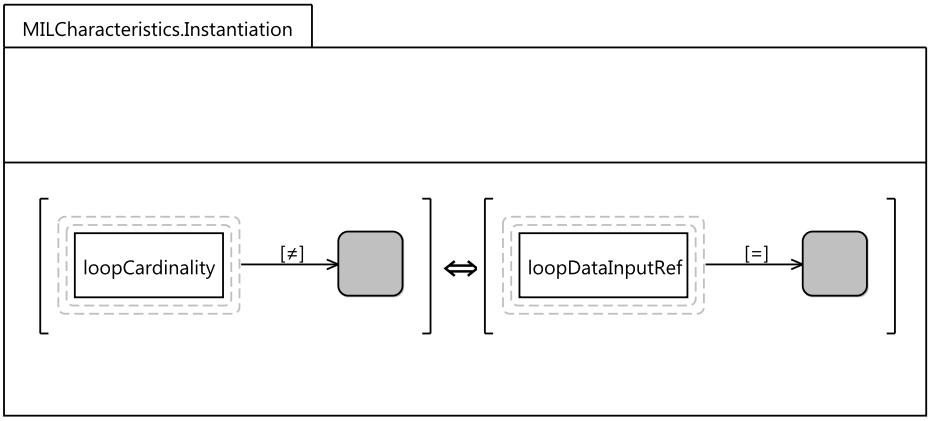


Figure B.104: The `Instantiation` assertion expressing invariants on the `MILCharacteristic`

B.30 The SubProcesses package

`SubProcesses` are commonly used to model `Activities` that need to be specified but should be hidden visually to not overcomplicate diagrams. As such, `SubProcess` has two subsets, one of which is `Transaction`, modelling atomic `Activities` that specify a protocol to guarantee atomicity. This constraint is expressed using the local invariant `SpecificMethods`. The specification introduces some ambiguity as that the class diagram proposes the `method` parameter be modelled as a string while the table of attributes specifies that it be of a "TransactionMethod" type. It has to be chosen to stick with the diagram and apply a constraint rather than introduce a new type.

The second subset of `SubProcess` is the `AdHocSubProcess` which defines not one but multiple `Activities` for which the execution is dependent on the `Activity's Performer` although an `ordering` might impose some constraints on the order of execution. A global invariant, `AdHocSubProcessRestrictions` satisfies the requirement that an `AdHocSubProcess` must contain an `Activity` but no `StartEvent` or `EndEvent`. The `OneStartEvent` global invariant ensures that a `SubProcess` can only have one `StartEvent` if it is indeed marked as being `triggeredByEvent`. This unique `StartEvent` is further constrained to be of a very specific `EventDefinition` as given by the custom set formed by a call to the *`EventDefinitions'`* `GetValidSubProcessStarts` global operation.

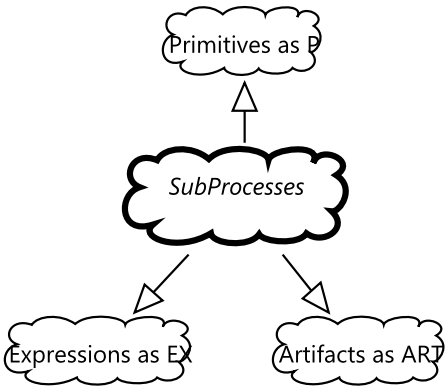


Figure B.105: The *`SubProcesses'`* package diagram

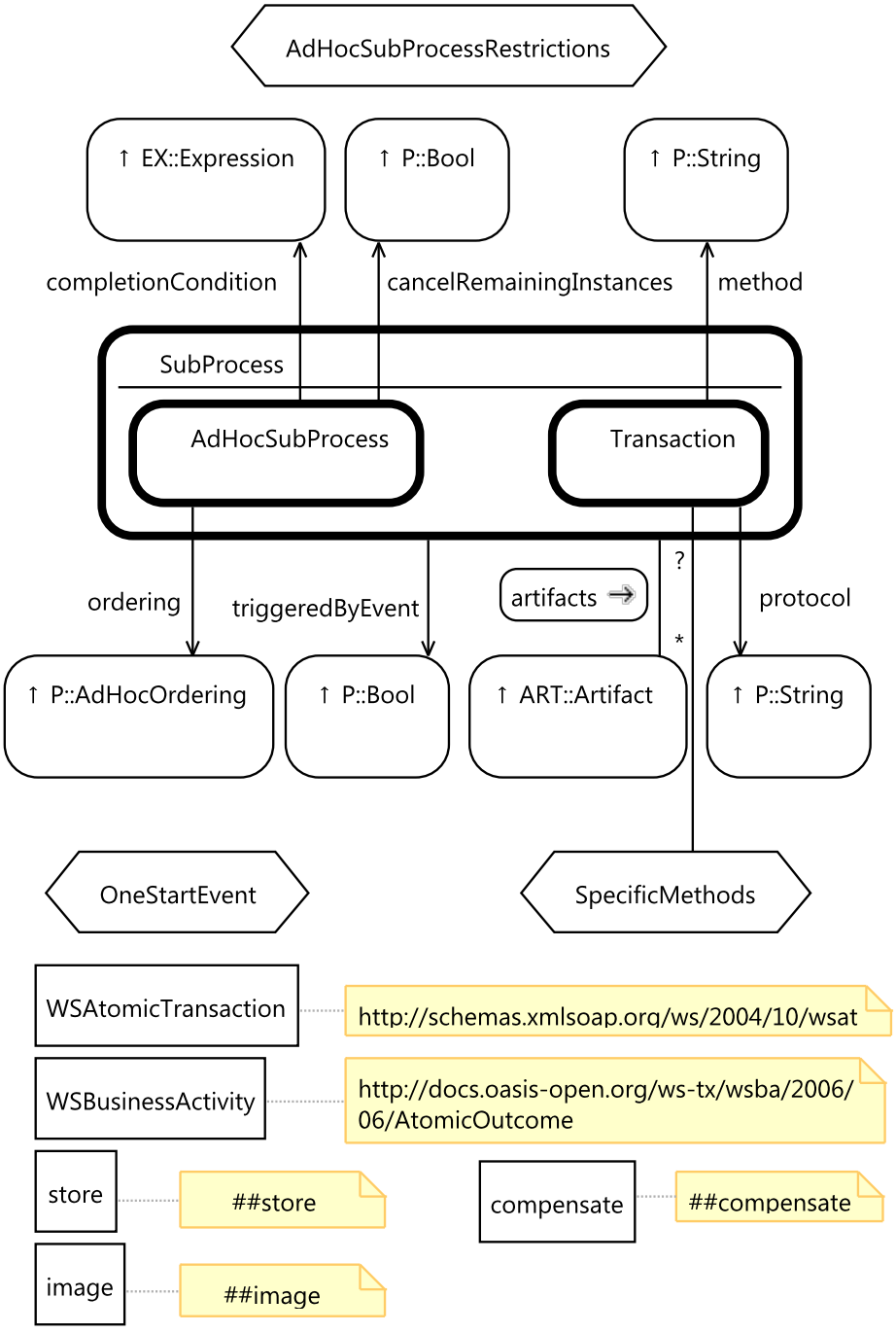


Figure B.106: The *SubProcesses'* structural diagram

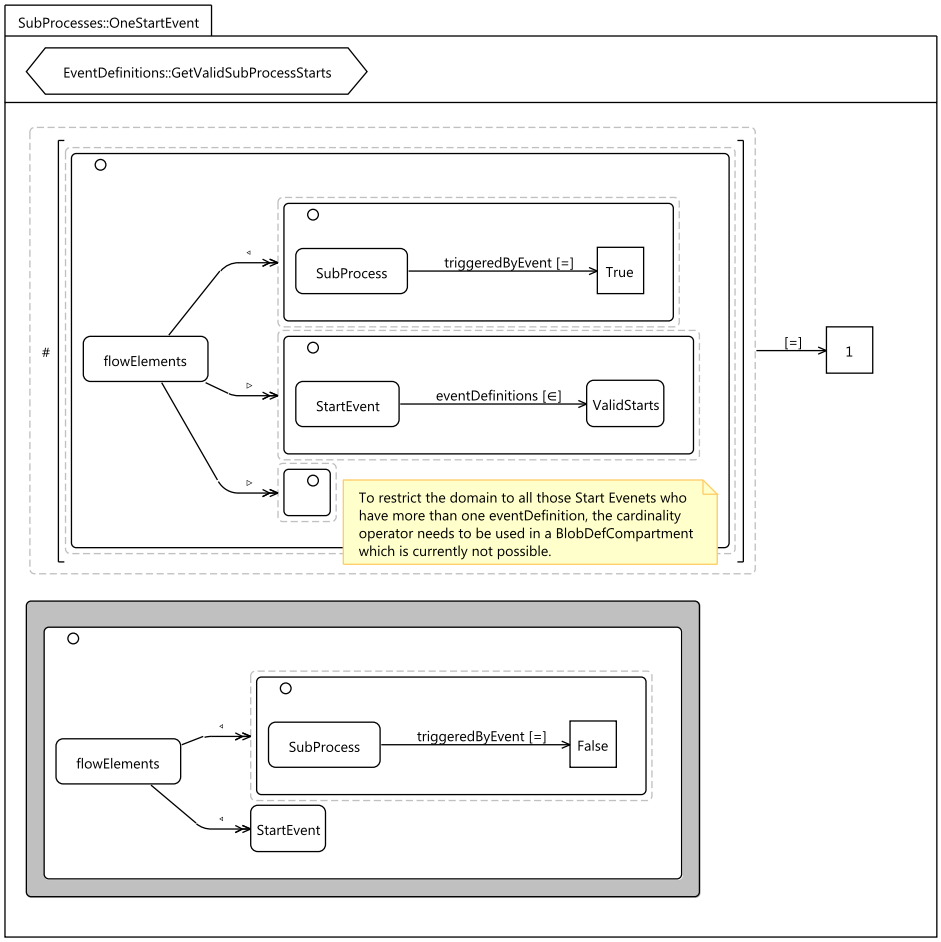


Figure B.107: The `OneStartEvent` assertion expressing invariants on the `SubProcess`

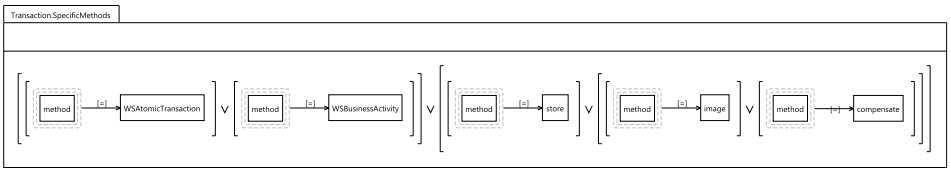


Figure B.108: The `SpecificMethods` assertion expressing invariants on the `Transaction`

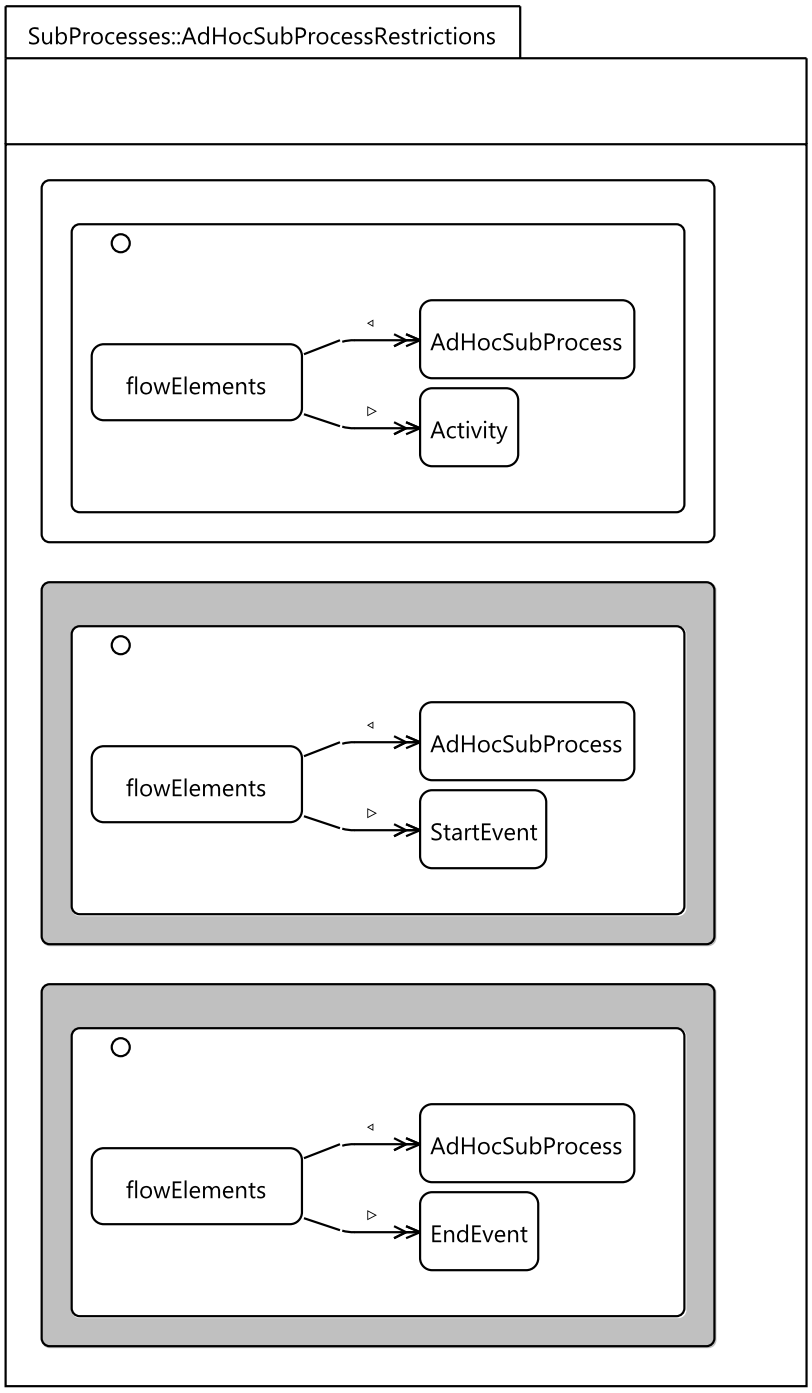


Figure B.109: The `AdHocSubProcessRestrictions` assertion expressing invariants on the `AdHocSubProcess`

B.31 The Activities package

They are defined by its subsets, `CallActivity`, `SubProcess` and `Task`. The VCL *`Activities`* package models this notion. `Activities` can handle `Data` by the `DataAssociations` and `InputOutputSpecifications` they can refer. As mentioned before, an `Activity` can loop as defined by its `LoopCharacteristic`. Moreover, it can be used as compensation, handling an `Event` with a `CompensationEventDefinition`. `Activities` also contain references to all `IntermediateEvenets` that are attached to its boundary.

A `CallActivity` allows an `Activity` to access globally defined `Processes` or `Globaltasks` from within the scope of the `Activity`. This makes those elements reusable. The package defines two constraints. The global `IOConstraints` satisfies the specification's requirement that the `CallActivity's ioSpecification` must exactly match that of the referenced `CallElement`. The local `FrameConditions` invariant specifies that the `startQuantity` and `CompletionQuantity` must always be at least 1 as is specified by the BPMN2 documentation. Similarly, the `Activity's Default` operation sets the default values for these properties.

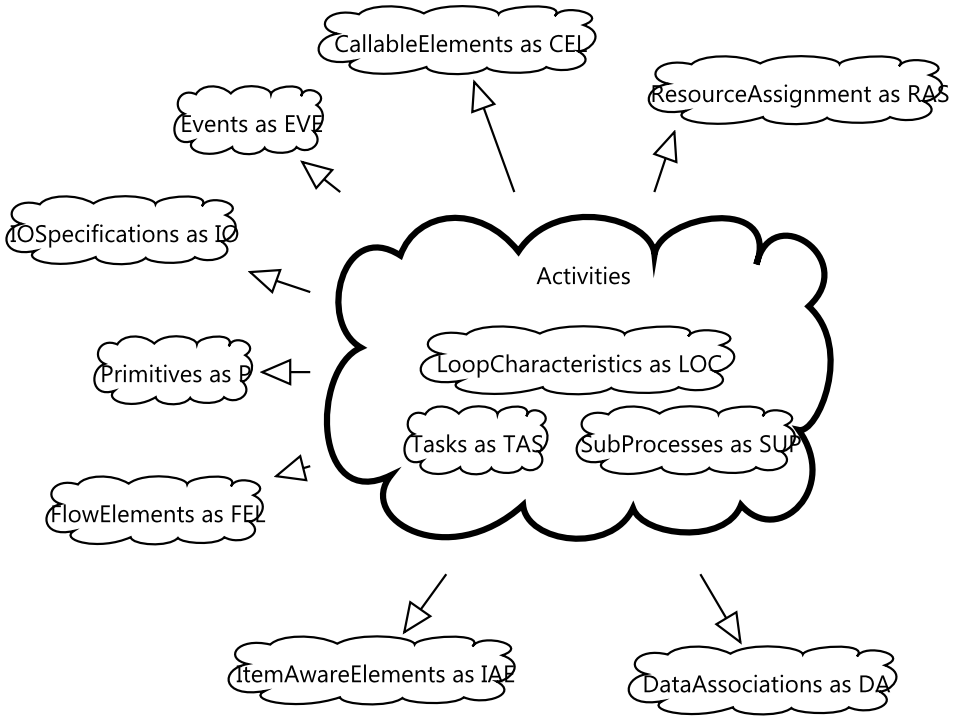


Figure B.110: The *`Activities'`* package diagram

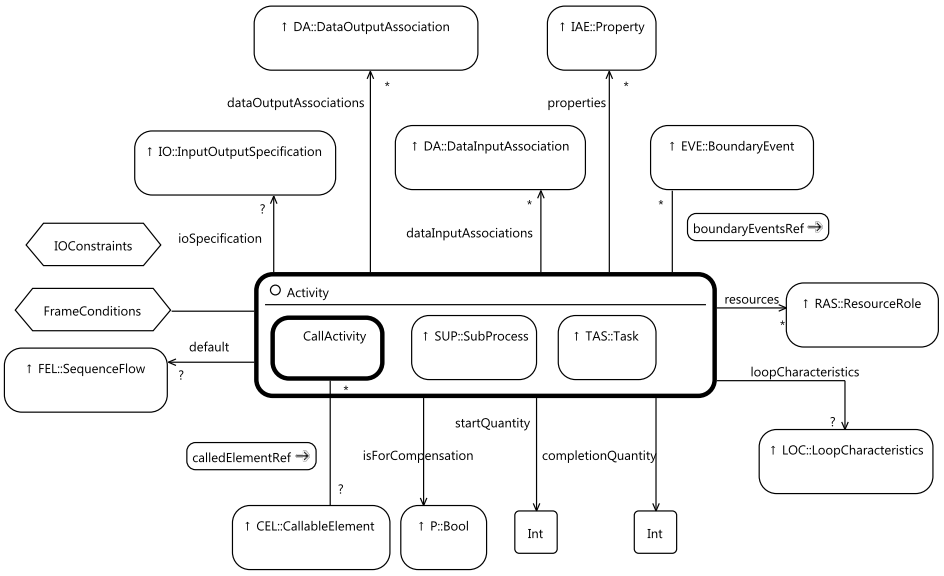


Figure B.111: The *Activities'* structural diagram

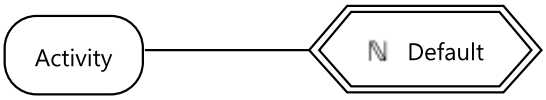


Figure B.112: The *Activities'* behaviour diagram

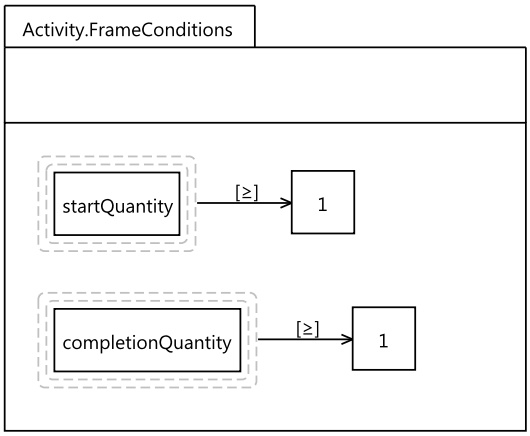


Figure B.113: The `FrameConditions` assertion expressing invariants on the `Activity`

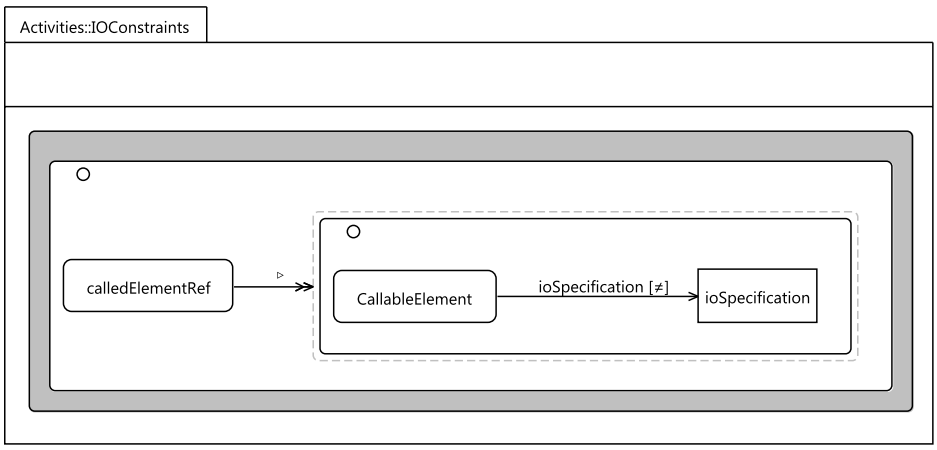


Figure B.114: The `IOConstraints` assertion expressing invariants on the `CallActivity`

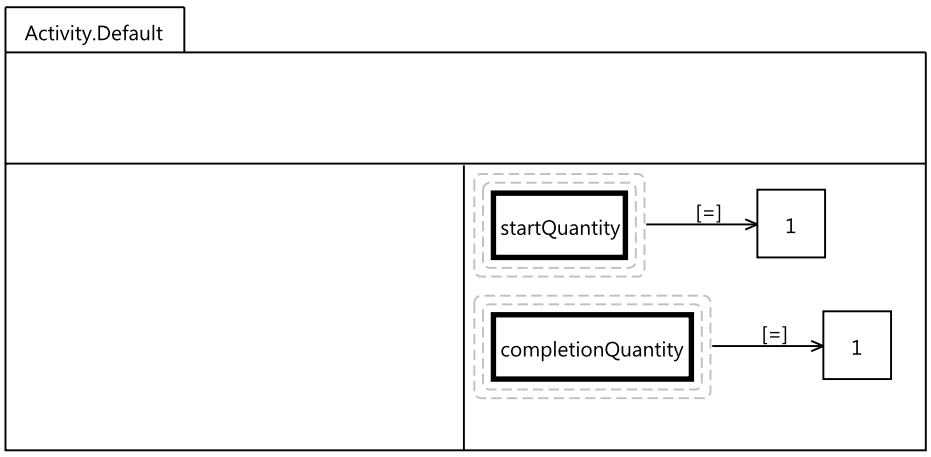


Figure B.115: The `Activity's Default` operation

B.32 The Gateways package

The *`Gateways`* VCL package groups the concept of `SequenceFlow` control elements. A `Gateway` is defined in VCL by its subsets: `InclusiveGateway`, `ExclusiveGateway`, `ComplexGateway`, `EventBasedGateway`, and `ParallelGateway`. In addition to the properties and relations inherited from `FlowElements`, `Gateway` specifies a `GatewayDirection`.

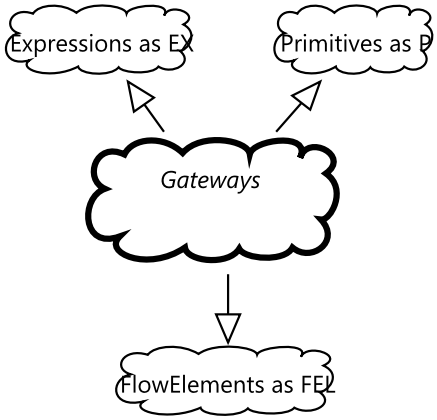


Figure B.116: The *Gateways'* package diagram

An `ExclusiveGateway` can have a `default SequenceFlow` which will be the path to follow should the `Expressions` on all other `SequenceFlows` fail to hold. While semantics are not covered by this document, it is worth mentioning that only the first path whose `Expressions` evaluates to true will be followed. Ideally, by design although no constraint exists, only one path's `Expression` should evaluate to true. An `InclusiveGateway` functions much like an `ExclusiveGateway` with the exception that all `Expressions` are evaluated and each `outgoing SequenceFlow` whose `Expressions` evaluated to true will be followed. The `InclusiveGateway` also makes use of a `default SequenceFlow`. The `Converging` behaviour may or may not synchronise `incoming SequenceFlows`.

A `ParallelGateway` is used to merge parallel `SequenceFlows` or to create them, independent of any `Expressions`. `ComplexGateways` are used to model more complex flow decisions than are offered by the `Exclusive-Gateway`. The `ComplexGateway` may carry an `activationCondition` in the form or an `Expression`. The `activationCount` holds at runtime the number of active `incoming SequenceFlows`. The `ComplexGateway` has an internal state, expressed by the property `waitingForStart` which is `True` if the `Gateway` is idle and `false` if it is waiting to be reset.

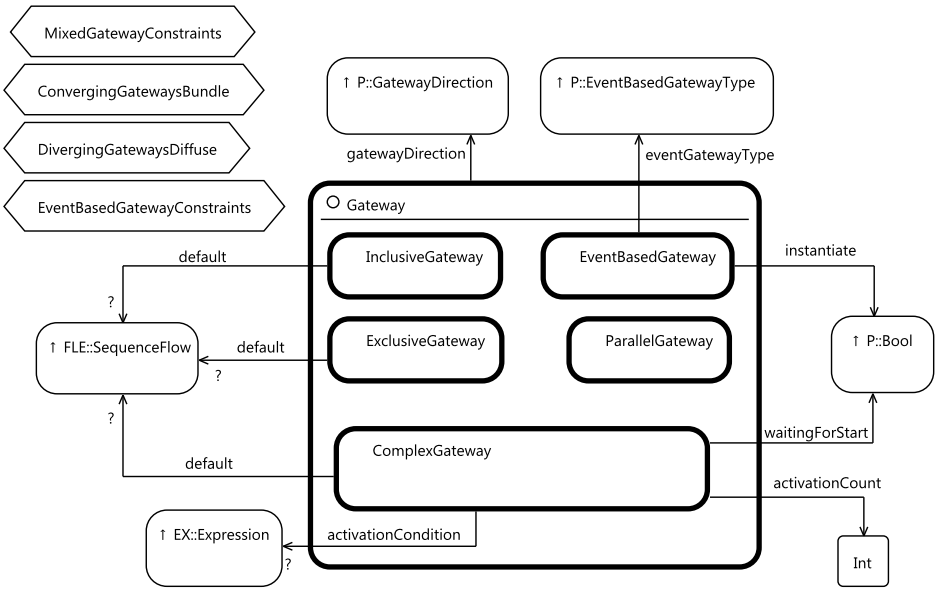


Figure B.117: The *Gateways'* structural diagram

An `EventBasedGateway` is used as a branching point with the decision about the branching relying on an `Event`. In order to make a decision, no outgoing `SequenceFlow` may include an `Expression` as modelled by the `EventBasedGatewayConstraints` invariant. The `outgoing SequenceFlows` will link to `Events` which will then, if triggered, take care of routing the flow onwards. The `EventBasedGateway` has two properties,`instantiate` and `eventGatewayType`. Both are used to enrich the `EventBasedGateway` and offer more uses. When the `instantiate` property is set to `True`, the `Gateway` is used as a starting point, conditionally starting one or more `Processes` depending on which `Events` are triggered. Should the `eventGatewayType` be set to `Parallel` the `Gateway` will still be active and allow for further events to be triggered, allowing multiple parallel flows stemming from one `Event`. A constraint on the type and instantiation properties is also satisfied by the previously mentioned invariant.

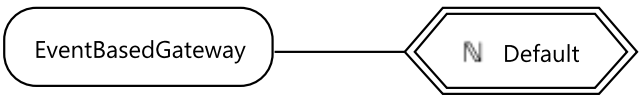


Figure B.118: The *Gateways'* behaviour diagram

The package specifies several invariants on the `SequenceFlow` towards and from `Gateways`. The `ConvergingGatewaysBundle` invariant satisfies the specification's requirements that `Gateway's` with a `GatewayType` of `Converging` must have multiple `incoming` `SequenceFlows` but no more than one `outgoing` flow. In VCL, this is accomplished by using the cardinality operator and using it to verify that in the set of tuples from `Gateways`, restricted to the ones with the correct property, to `SequenceFlow` there is more than one, respectively at most one element. Similar invariants exist for diverging and mixed `Gateways`, the `DivergingGatewaysDiffuse` and `MixedGatewayConstraints` invariants. The `EventBasedGateway's Default` operation sets a default value as required by the specification.

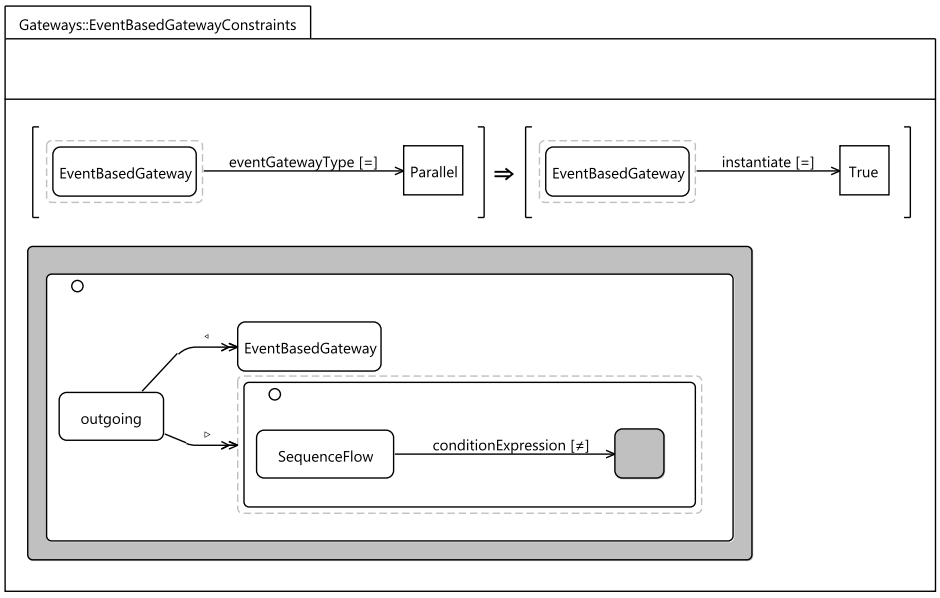


Figure B.119: The `EventBasedGatewayConstraints` assertion expressing constraints on the gateway

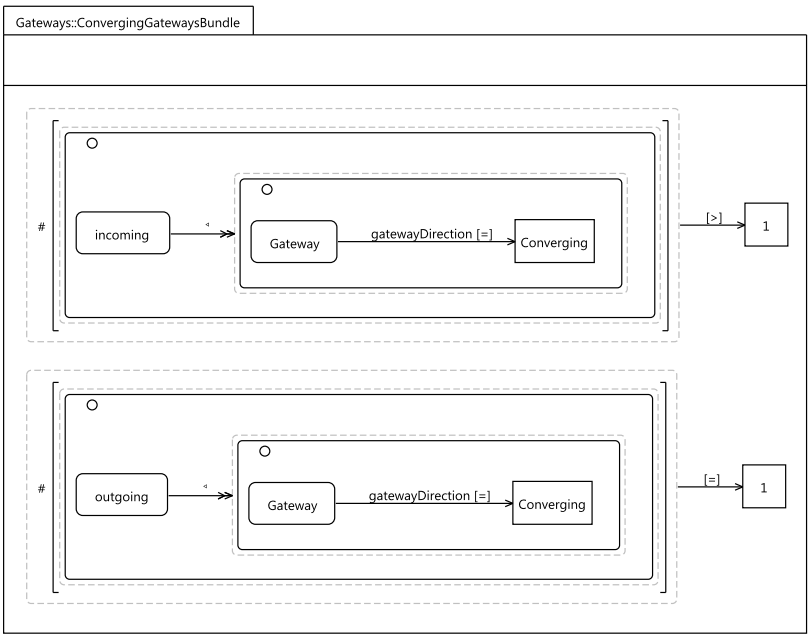


Figure B.120: The `ConvergingGatewaysBundle` assertion expressing constraints on the gateway

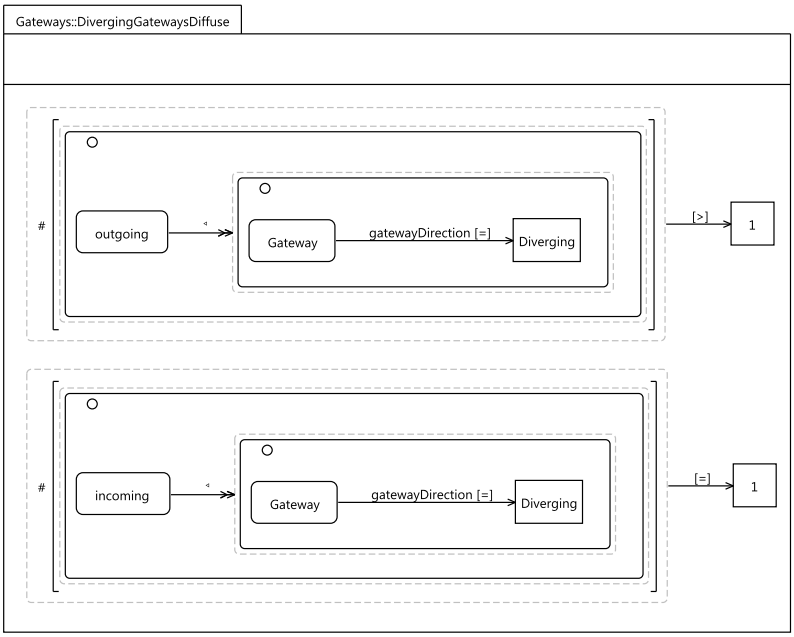


Figure B.121: The `DivergingGatewaysDiffuse` assertion expressing constraints on the gateway

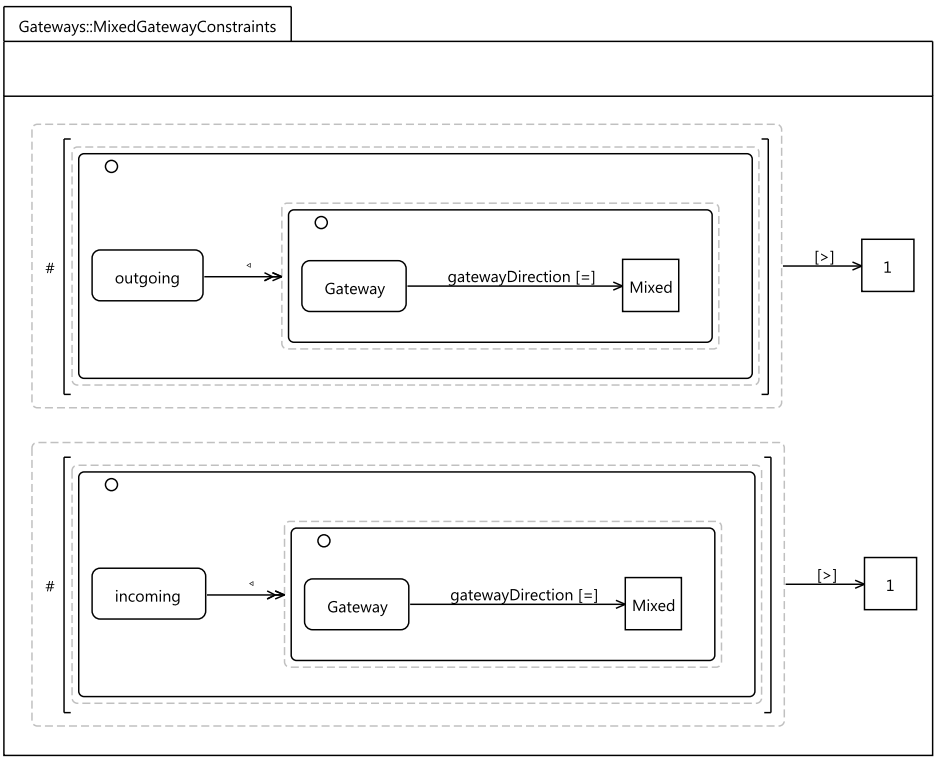


Figure B.122: The `MixedGatewayConstraints` assertion expressing constraints on the gateway

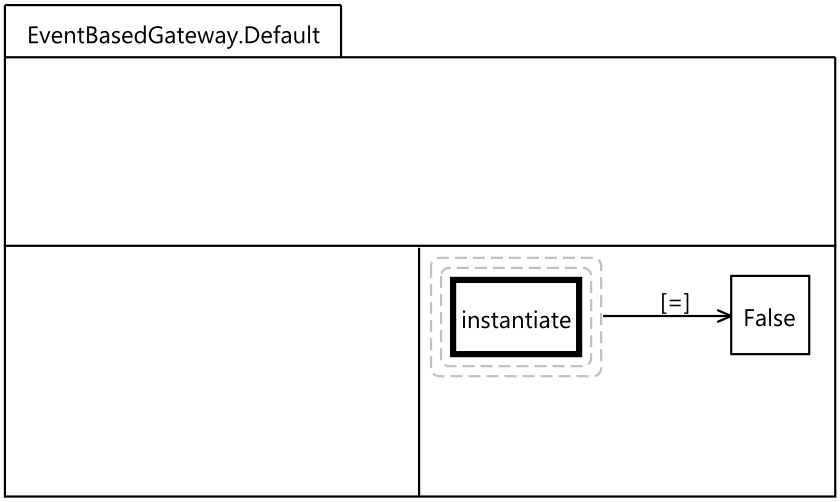


Figure B.123: The `EventBasedGateway's Default` operation

B.33 The Tasks package

All of the `Task's` subsets as depicted on the SD but `ManualTask` and `ScriptTask` specify a technology used to complete the task. They have a `Default` operation included in the VCL `Tasks` package which models these concepts. `ReceiveTask` and `SendTask` include references to the `Message` being send respectively received as well as the `Operation` invoked or invoking the `Task`. The `ServiceTaskMessageConstraints` invariant satisfies the requirements that the `DataOutput`, if present. must match the `ItemDefinition` of the `Message` the `ServiceTask` will send. The `ReceiveTask` has a similar constraint which has not been modelled due to redundancy.

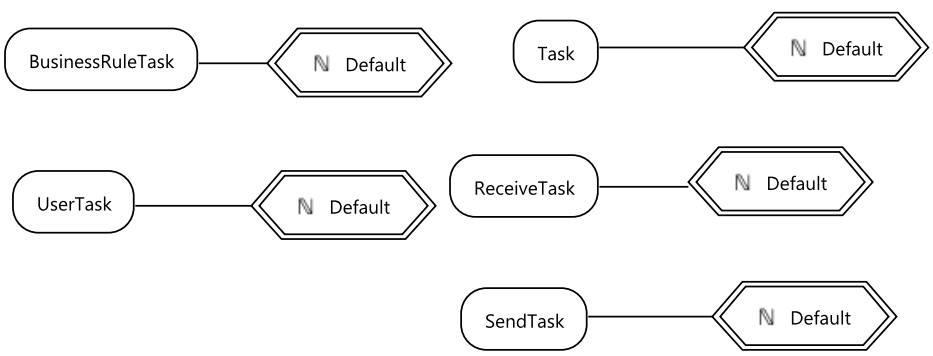


Figure B.124: The *`Tasks'`* package diagram

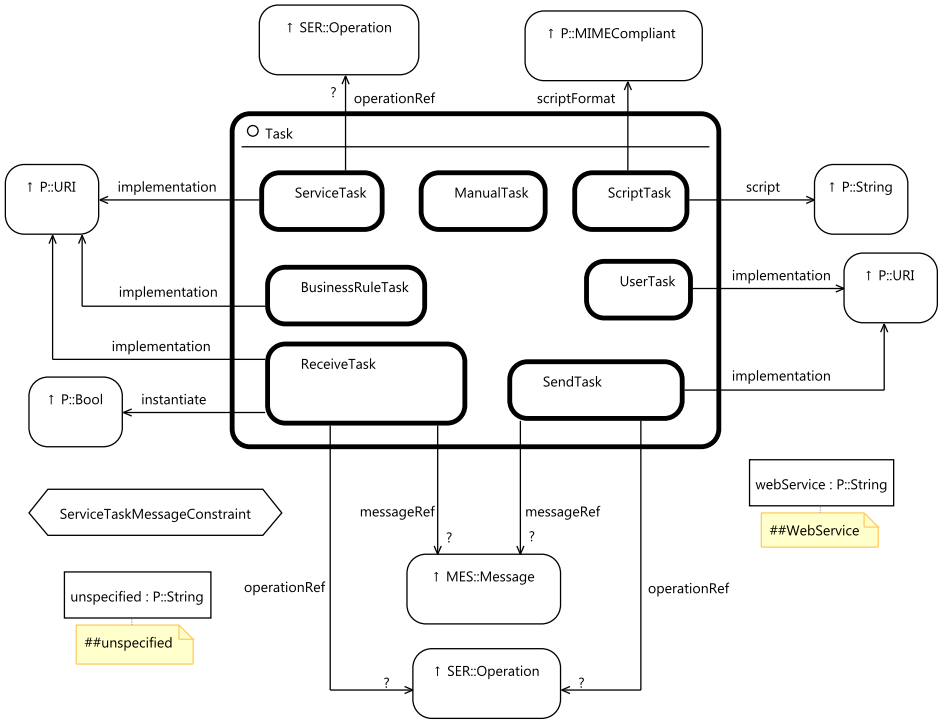


Figure B.125: The *Tasks'* structural diagram

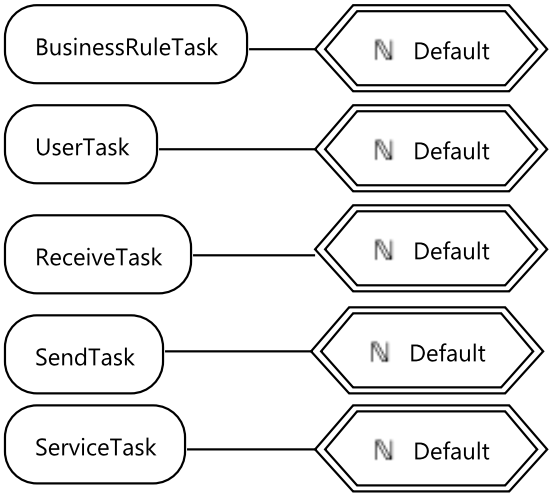


Figure B.126: The *Tasks'* behaviour diagram

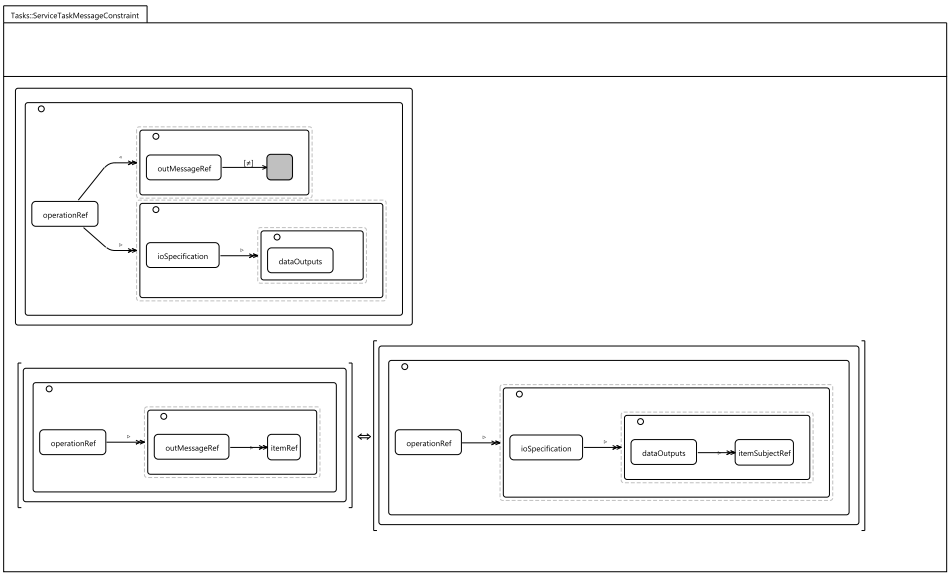


Figure B.127: The `ServiceTaskMessageConstraint` assertion expressing invariants on the `ServiceTask's` messaging facilities

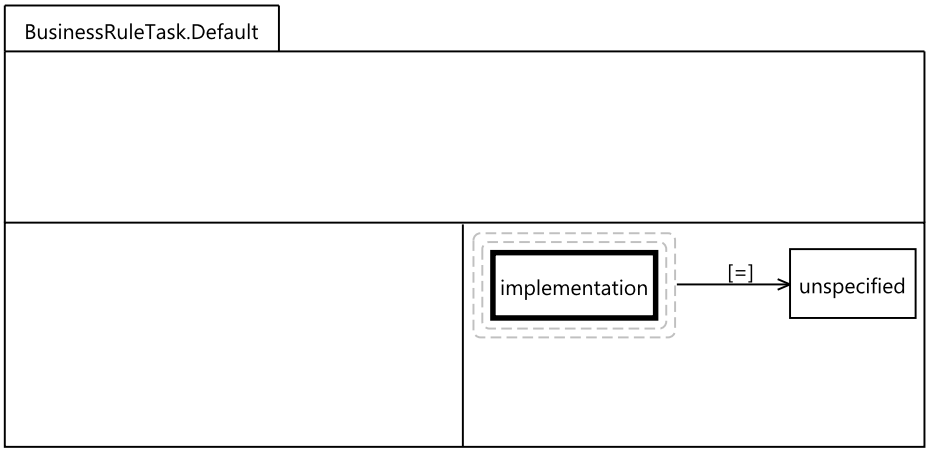


Figure B.128: The `BusinessRuleTask's Default` operation

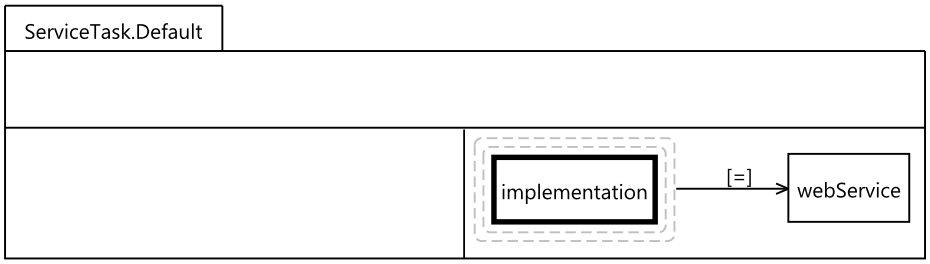


Figure B.129: The `ServiceTask's Default` operation

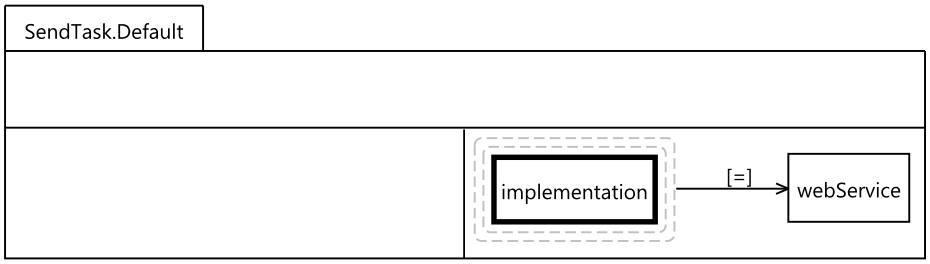


Figure B.130: The `SendTask's Default` operation

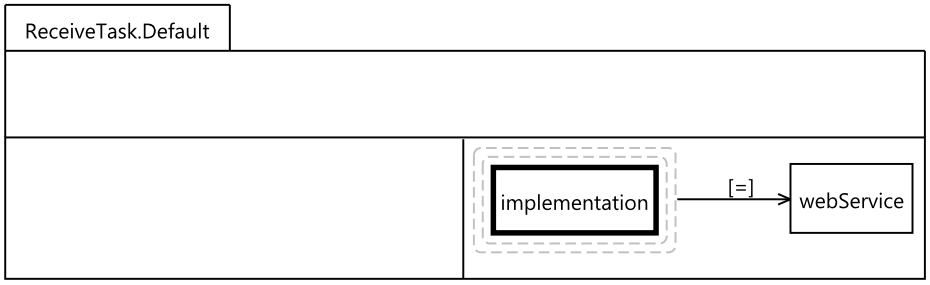


Figure B.131: The `ReceiveTask's Default` operation

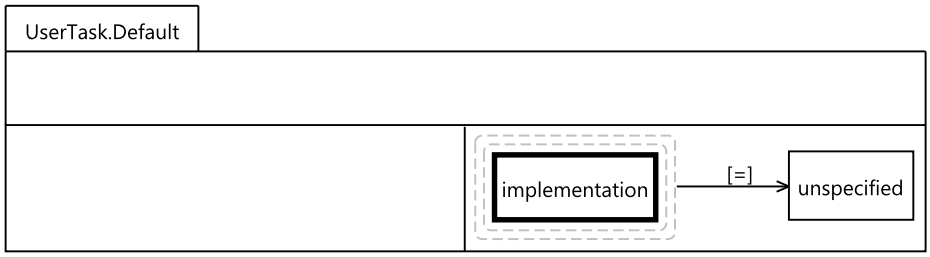


Figure B.132: The `UserTask's Default` operation

B.34 The Participants package

The `Participants' name` can be substituted for the `name` of the `PartnerEntity` or `PartnerRole` or combined with them. The `Participant` holds a reference to its `Process`, enabling the formulation of the `GetParticipantGivenProcess` assertion which retrieves a single `Participant` from the tuples given by the relation and a inputted `Process`. The `Participant` also holds also references to its `PartnerRole` and `PartnerEntity`. A `Participant` can be implemented remotely in which case an `endPointRefs` relation holds the `EndPoint` at which the implementation can be accessed.

The `ParticipantMultiplicity` has two properties, `minimum` and `maximum`, both represented by IntBlobs in VCL, specifying the multiplicity. These bounds specify a range which the actual number of `Participants` may vary in. The specification mentions a runtime attribute `numParticpants`. It is not modelled by the specification's diagrams and has therefore not been modelled here as well.

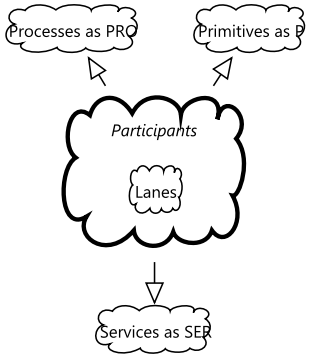


Figure B.133: The *`Participants'`* package diagram

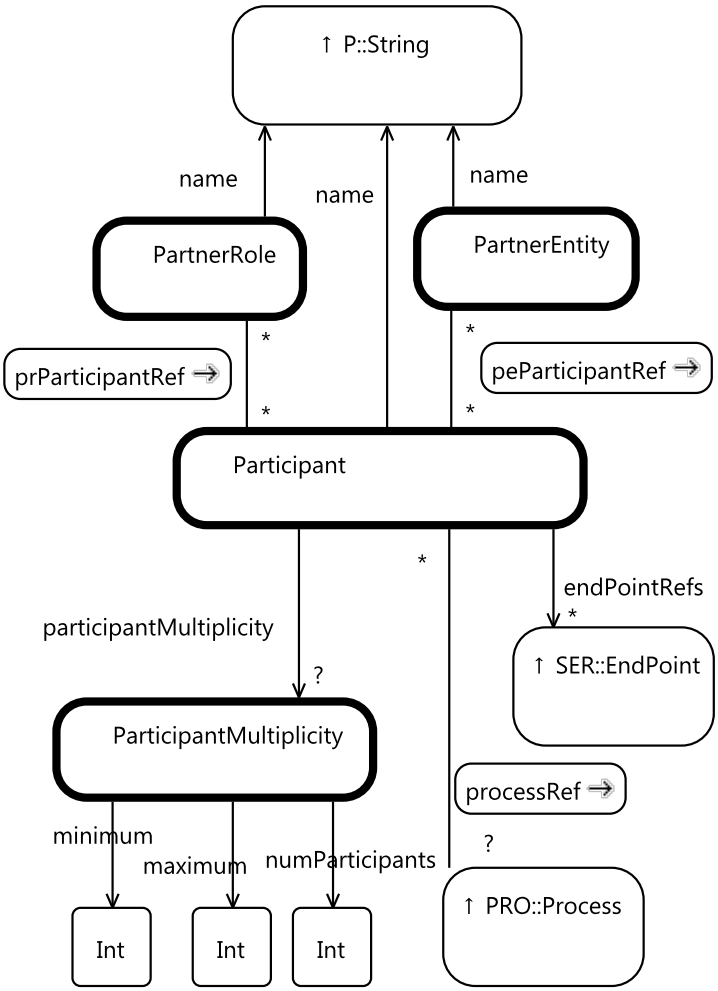


Figure B.134: The *Participants'* structural diagram

Figure B.135: The *Participants'* behaviour diagram

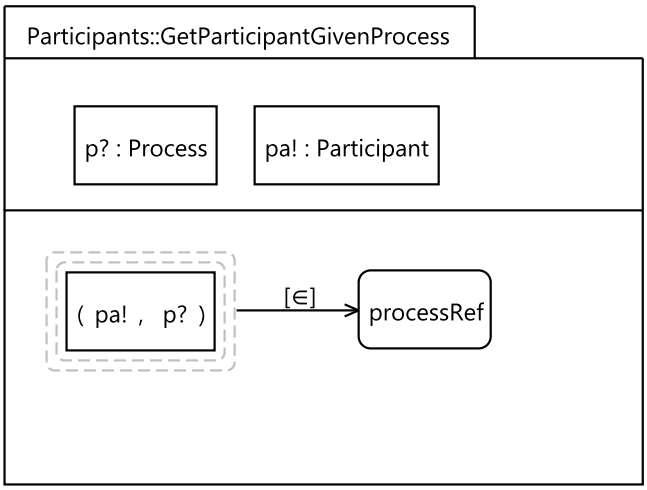


Figure B.136: The *Tasks'* `GetParticipantGivenProcess` operation

B.35 The Core package

The *Core* VCL ensemble package is an analogy to the BPMN2 core package. It groups all BPMN2 concepts from the metamodels. Due to nesting, concepts are already grouped in packages such as *Foundation* and *Common*. Several packages need to merge concepts such as *Infrastructures* and *Foundation* which merger `RootElement` and `BaseElement`. Merging is a must as the two merged concepts are defined in both packages. Ideally, they would have only been defined in the *Foundation* package but this would have meant circular imports which are not allowed in VCL but can be done in UML which is what the BPMN2 diagrams are specified in.

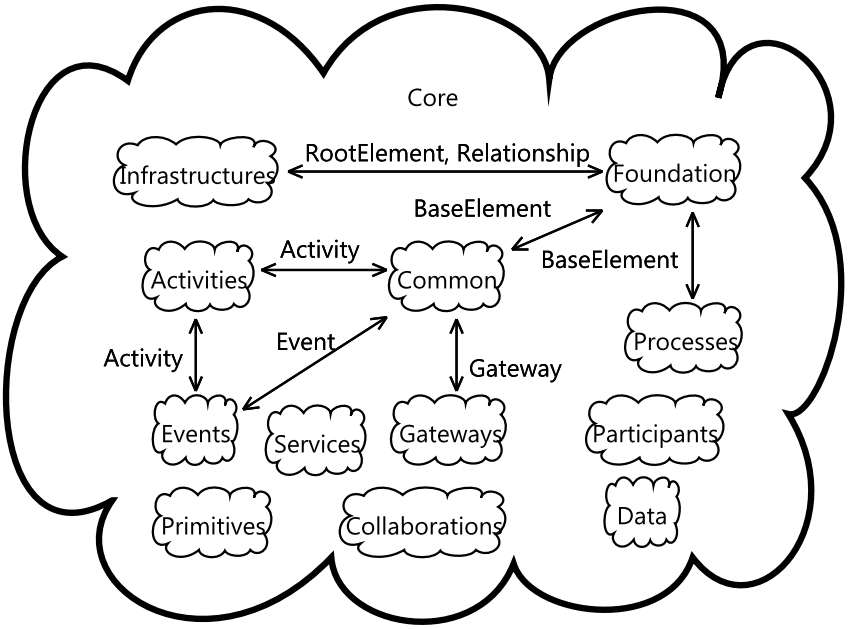


Figure B.137: The *Core's* package diagram

Appendix C

BusinessToTable metamodel VCL packages

The following sections present all Visual Contract Language (VCL) packages needed to build the Business-to-Table (BtT) metamodel as given by its specification document [23]. The packages will detail all diagrams. Each section will provide an overview of the package and complement the general description given in Section 5.5.

For an introduction on VCL, please consider reading Section 4. In this appendix, the following notation is used; Structural Diagrams (SD), Behavioural Diagrams (BD), Package Diagrams (PD), Assertion Diagrams (AD), Contract Diagrams (CD). Table B.1 lists all VCL packages and the corresponding figures for each VCL package.

Table C.1: BTT metamodel VCL packages and figures

Package name	Figures
Mutator C.1	PD C.1 SD C.2
BusinessToTable C.2	PD C.3 SD C.4

C.1 The Mutator package

The *Mutator* package imports all BPMN2 metamodel packages containing elements that are immediately visualised and thereby can be actively manipulated by the users. These elements are: `Lanes`, `Collaborations`, `Glob-`

alTasks, FlowElements, Artifacts, Messages, FlowElementcontainers, MessageFlows, and Participants. The union of these sets forms the Subject.

A Subject has Attributes identified by their name name and value. Additionally, a Subject can hold References and in turn the target of a Reference.

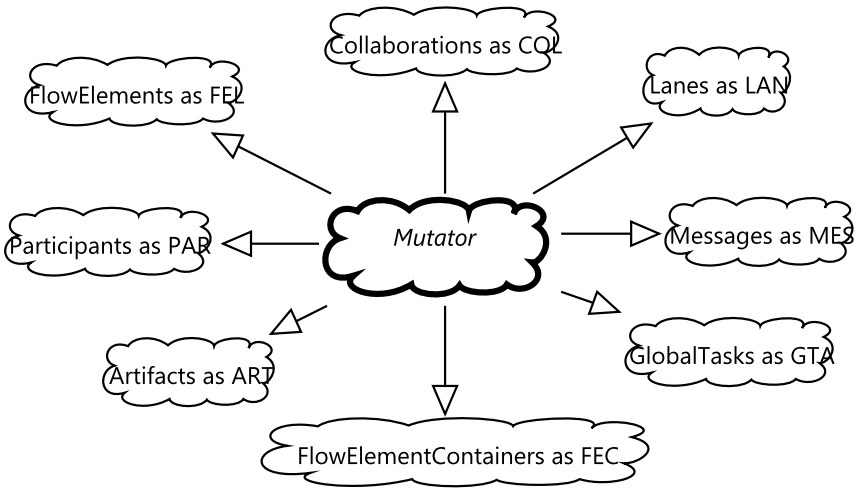


Figure C.1: The *Mutator's* package diagram

C.2 The BusinessToTable package

The *BusinessToTable* package imports the *TUI* package in order to access the Zones. The package introduces the Packet concept, a message that is sent by the Zone, eventually stored in a PackageStore, before being analysed by an Interpreter. The latter then triggers one or more Happenings which are defined by the subsets: Create, Destroy, ChangePos, or TextInput. These are all the interactions that the prototype, as designed by Bicheler in the scope of his Master's Thesis [104], requires. More Happenings can be added if needed. Happenings impact a Subject as defined in the *Mutator* package.

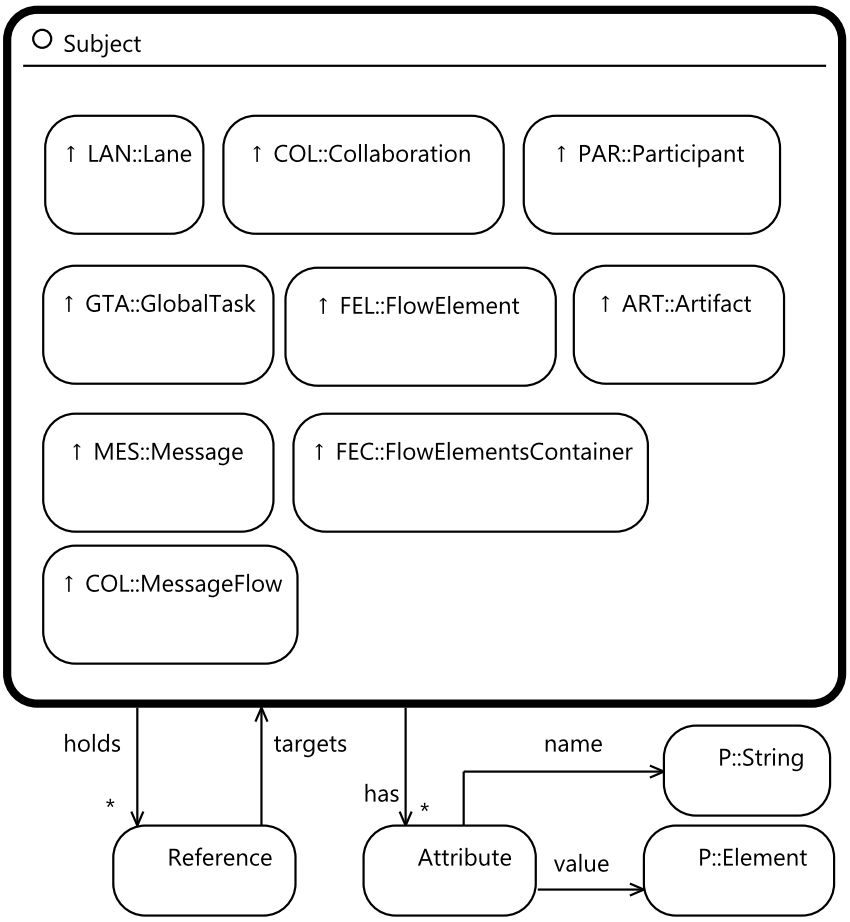


Figure C.2: The *Mutator's* structure diagram

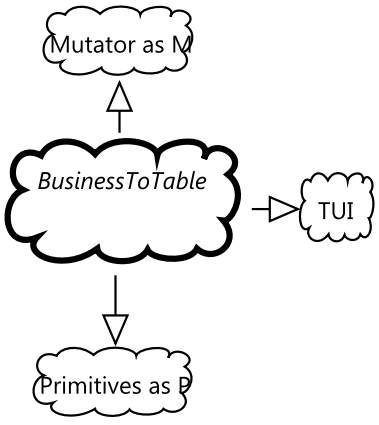


Figure C.3: The *BusinesssToTable's* package diagram

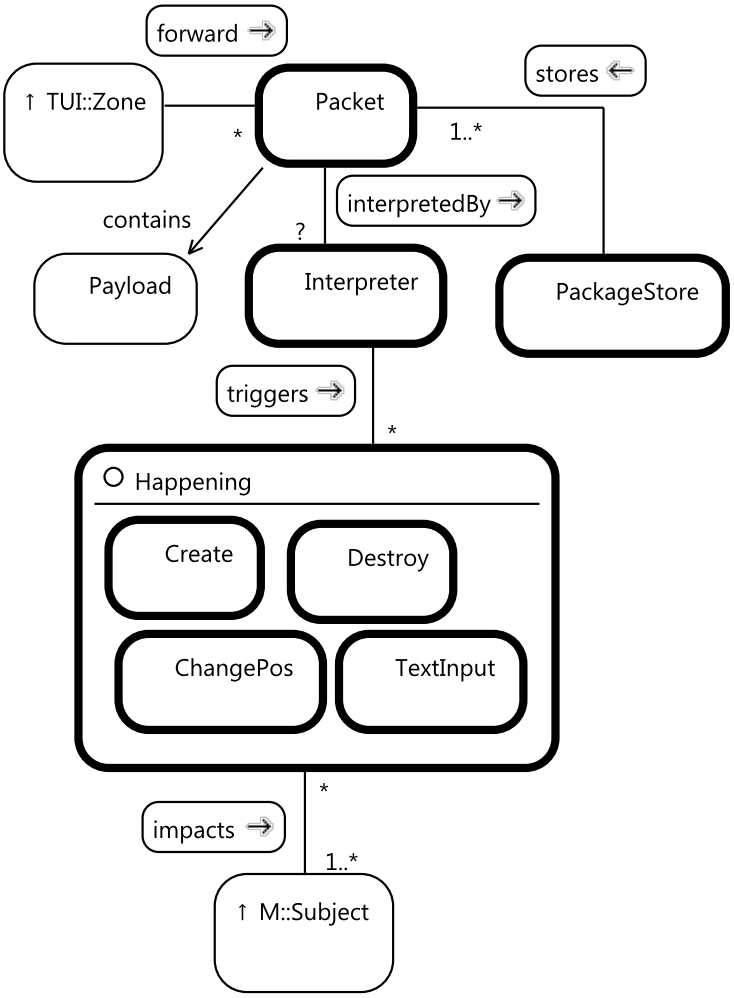


Figure C.4: The *BusinesstoTable's* structure diagram

Appendix D

Ideation scenario VCL packages

The following sections present all Visual Contract Language (VCL) packages needed to build the ideation model as specified for the ideation scenario used in Chapter 6. The following section will provide an overview of the package and complement the general description given in Section 5.7.

For an introduction on VCL, please consider reading Section 4. In this appendix, the following notation is used; Structural Diagrams (SD), Behavioural Diagrams (BD), Package Diagrams (PD), Assertion Diagrams (AD), Contract Diagrams (CD). Table D.1 lists all VCL packages and the corresponding figures for each VCL package.

Table D.1: Ideation model VCL packages and figures

Package name	Figures
Ideation D.1	PD D.1 SD D.2

D.1 The Ideation package

The *Ideation* package contains the formal VCL model of the BPMN2 scenario shown in Figure 6.1. The SD displays the concept of `IdeationScenario` containing one pool, `IdeationLane` which contains three lanes; `BackOffice`, `FrontOffice`, and `Ideator`. Each of those packs a `name` and an `id`. The `Ideator` lane includes a process composed of several BPMN2 flow nodes. Each of those has an `id` and all activities have a `name`. All

flow nodes are connected through sequence flows such as `SourceFlow` or `CallFlow`. These always hold references to sources and targets of the flow. As the flow nodes also hold references to the sequence flows, this induces a large number of relations which makes the model a bit hard to read.

The `IssueOpinion`, `UseCommFeature`, and `Contribute` activities hold a `loopCharacteristics` modelled by the `MultiInstanceMarker`. This defines properties to keep track of the number of instances and their states by counters. The `MultiInstanceMarker` instantiates the *`LoopCharacteristics`* package's `MILCharacteristic`. However, there is a discrepancy between the models in that the BPMN2 metamodel does not include the runtime attributes depicted on the model. While the specification mentions these attributes, none of the diagrams included them. Hence, either they should have been included on the diagrams in the specification or a runtime attribute concept should have been defined which would have allowed to define any number of runtime attributes on the model level.

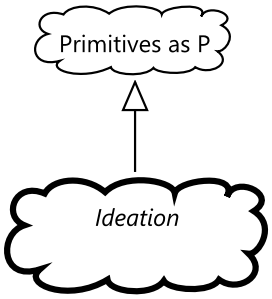


Figure D.1: The *`Ideation's`* package diagram

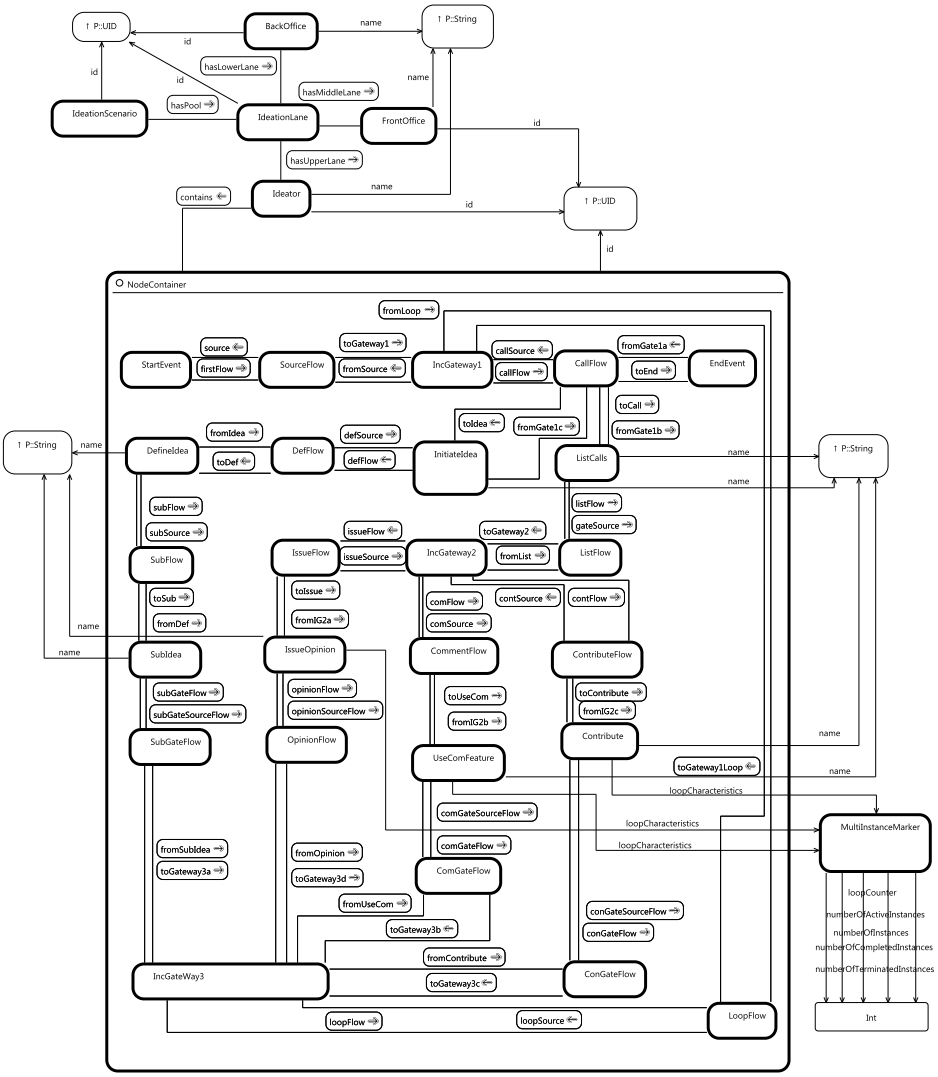


Figure D.2: The *Ideation's* structure diagram

Appendix E

Widget model VCL packages

The following sections present all Visual Contract Language (VCL) packages needed to build the widget model as given by the prototype widgets designed using TWT in the scope of Bicheler's Master's Thesis [104]. The following packages will detail all implemented diagrams. Each section will provide an overview of the package and complement the general description given in Section 5.8.

For an introduction on VCL, please consider reading Section 4. In this appendix, the following notation is used; Structural Diagrams (SD), Behavioural Diagrams (BD), Package Diagrams (PD), Assertion Diagrams (AD), Contract Diagrams (CD). Table E.1 lists all VCL packages and the corresponding figures for each VCL package.

Table E.1: Widget model VCL packages and figures

Package name	Figures
Prototype E.1	PD E.1 SD E.2
ZoomBehaviour E.2	PD E.3 SD E.4

E.1 The Prototype package

The *Prototype* package models the concept of `WidgetLayer` and that it contains all four widgets defined by the *Prototype*, `Zoom`, `Stamp`, `Link`, and `AnnoMarker`. All of these widgets hold an `IDAttribute` and at least one handle. The `Link` even holds two handles, `Arrow` and `Tail`. All but the `AnnoMarker` also hold a visual component, such as for example the `Imprint` held by the `Stamp`. Each widgets holds an `identity` that defines its behaviour. Those identities are defined in the corresponding packages, however, due to problems explained in Section 5.8, only the *ZoomBehaviour* package has been defined. Nevertheless, all of those remotely defined concepts are specified here, making the structural diagram complete.

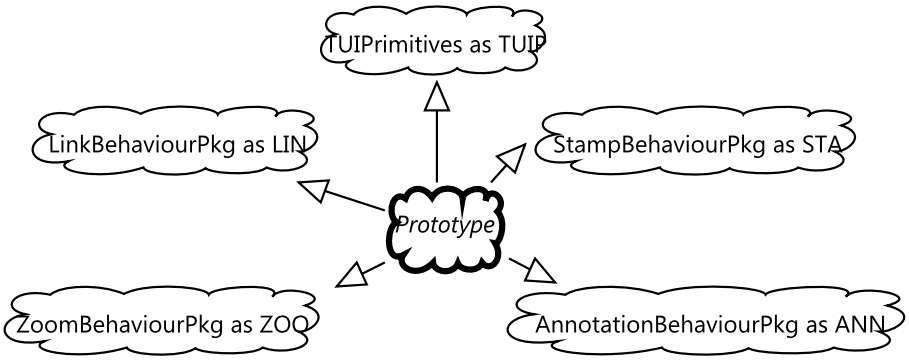


Figure E.1: The *Prototype's* package diagram

E.2 The ZoomBehaviour package

The *ZoomBehaviour* package models the identity of a `Zoom` widget used in the ideation scenario. The identity holds three functions, `DropBind`, `LiftUnbind`, and `RotateZoom`. Due to problems detailed in Section 5.8, this model is only partially implemented. In the final implementation, all of the functions would have been defined remotely in their own packages, specifying all mappings of attributes, actions and effects that would have defined the function. This would have made it possible to reuse functions that are likely recurring, such as `DropBind`.

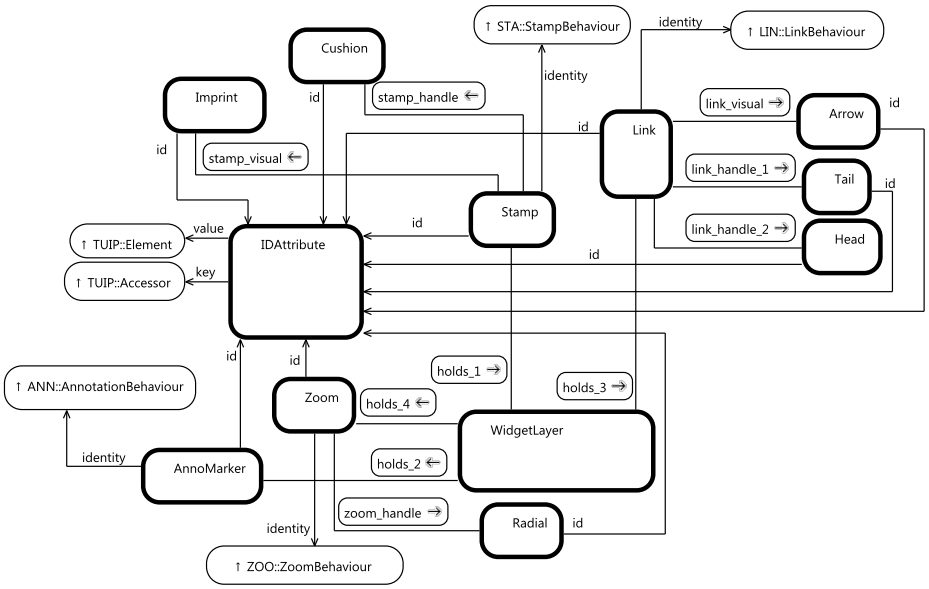


Figure E.2: The *Prototype's* structure diagram

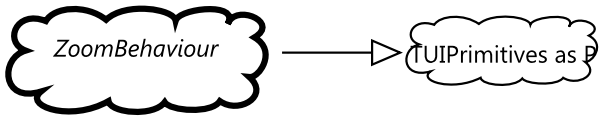


Figure E.3: The *ZoomBehaviour's* package diagram

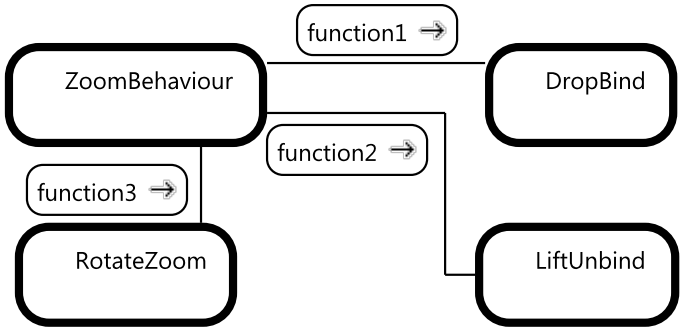


Figure E.4: The *ZoomBehaviour's* structure diagram

Appendix F

List of literature resources

- Amlio, N., & Kelsen, P. (2010). Modular design by contract visually and formally using VCL. Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on (pp. 227234). IEEE. doi:10.1109/VLHCC.2010.39

- Amlio, N., Glodt, C., & Kelsen, P. (2011). Building VCL Models and Automatically Generating Z Specifications from Them. (M. Butler & W. Schulte, Eds.)FM 2011: Formal Methods, 6664, 149-153. Springer Berlin Heidelberg. doi:10.1007/978-3-642-21437-0

- Amio, N., Kelsen, P., Ma, Q., & Glodt, C. (2010). Using VCL as an aspectoriented approach to requirements modelling. Transactions on Aspect Oriented Software Development, 7, 151-199. doi:10.1007/978-3-642-16086-8_5

- Bardohl, R., Ehrig, H., De Lara, J., & Taentzer, G. (2004). Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. Fundamental Approaches to Software Engineering, 214228. Springer.

- Bottoni, P., Koch, M., Parisi-presicce, F., & Taentzer, G. (2001). A Visualization of OCL using Collaborations. (M. Gogolla & C. Kobryn, Eds.)UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, 2185, 257-271. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/3-540-45441-1

- Brockmans, S., Volz, R., & Eberhart, A. (2004). Visual modeling of OWL DL ontologies using UML. The Semantic WebISWC 2004 (pp. 198213). Springer.

- Burnett, I., Baker, M. J., Bohus, C., Carlson, P., Yang, S., & Van Zee, P. (1995). Scaling Up Visual Programming Languages. Computer, 28(3), 45-54. doi:10.1109/2.366157

- Burnett, M., Atwood, J., Walpole Djang, R., Reichwein, J., Gottfried, H., & Yang, S. (2001). Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. Journal of functional programming, 11(2), 155206. Cambridge University Press.

- Chailloux, E., & Codognet, P. (1997). Toward visual constraint programming. Visual Languages, 1997. Proceedings. 1997 IEEE Symposium on, 420-421.

- Genon, N., Amyot, D., & Heymans, P. (2011). Analysing the Cognitive Effectiveness of the UCM Visual Notation. System Analysis and Modeling: About Models, 6598/2011, 221240. Springer. Retrieved from http://www.springerlink.com/index/G7746GM683W97620.pdf

- Gil, J. Y., Howse, J., & Kent, S. (1999). Constraint Diagrams : A Step Beyond UML. Technology of Object-Oriented Languages and Systems (TOOLS USA99). Santa Barbara, California , USA. Retrieved from http://kar.kent.ac.uk/id/eprint/21740

- Halpin, T. (1998). Object-Role Modeling: an overview.

- Howse, J., Schuman, S., Stapleton, G., & Oliver, I. (2009). Diagrammatic Formal Specification of a Configuration Control Platform. Electronic Notes in Theoretical Computer Science, 259, 87-104. doi:10.1016-/j.entcs.2009.12.019

- Kent, S. (1997). Constraint diagrams: visualizing invariants in object-oriented models. OOPSLA 97 Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (Vol. 32). ACM.

- Lin, J., Thomsen, M., & Landay, J. A. (2002). A visual language for sketching large and complex interactive designs. Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves (pp. 307314). New York, New York, USA: ACM. doi:10.1145/503429.503431

- Lohmann, M., Sauer, S., & Engels, G. (2005). Executable Visual Contracts. 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC05) (pp. 63-70). IEEE. doi:10.1109/VLH-CC.2005.35

- Lucanin, D., & Fabek, I. (2011). A visual programming language for drawing and executing flowcharts. MIPRO, 2011 Proceedings of the 34th International Convention (pp. 16791684). IEEE.

- Muetzelfeldt, R., & Massheder, J. (2003). The Simile visual modelling environment. European Journal of Agronomy, 18(3-4), 345-358. doi:10.1016/S1161-0301(02)00112-0

- Group, O. M. (2006). Object Constraint Language - OMG Available Specification - Version 2. OMG.

- Sadi, S., & Maes, P. (2007). subTextile: Reduced event-oriented programming system for sensate actuated materials. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007), 171-174. Ieee. doi:10.1109/VLHCC.2007.37

- Schmidt, A., & Varr, D. (2003). CheckVML: A tool for model checking visual modeling languages. UML 2003 - The Unified Modeling Language. Modeling Languages and Applications, 2863, 9295. Springer.

- Schurr, A., Winter, A., & Zundorf, A. (1995). Visual programming with graph rewriting systems. Visual Languages, Proceedings., 11th IEEE International Symposium on (pp. 326333). Darmstadt , Germany: IEEE.

- Spratt, L., & Ambler, A. (1993). A visual logic programming language based on sets and partitioning constraints. 993 IEEE Symposium on Visual Languages (pp. 204-208). IEEE Comput. Soc. Press. doi:10.1109/VL.1993.269597

- Sprinkle, J., & Karsai, G. (2004). A domain-specific visual language for domain model evolution. Journal of Visual Languages & Computing, 15(3-4), 291-307. doi:10.1016/j.jvlc.2004.01.006

- Stapleton, G. (2005). A Decidable Constraint Diagram Reasoning System. Journal of Logic and Computation, 15(6), 975-1008. doi:10.1093/logcom/exi041

- Swenson, K. (1993). A visual language to describe collaborative work. Visual Languages, 1993., Proceedings 1993 IEEE Symposium on (pp. 298-303). Bergen: IEEE.

- Varro, D. (2003). Towards Symbolic Analysis of Visual Modeling Languages. Electronic Notes in Theoretical Computer Science, 72(3), 51-64. doi:10.1016/S1571-0661(04)80611-X

- Visual Contract Language.
  Retrieved March 7, 2012, from http://vcl.gforge.uni.lu/

- Visual OCL.
  Retrieved March 6, 2012, from http://tfs.cs.tu-berlin.de/vocl/

- VENSIM
  Retrieved March 5, 2012, from http://www.vensim.com/
- AnyLogic
  Retrieved March 5, 2012, from http://www.xjtek.com/

Appendix G

Study scenario requirements

G.1 BPMN2 requirements

G.1.1 Structural requirements

- **BS1** – The model diagram, subsequently model, holds a Collaboration or Process.
- **BS2** – The model has Events.
- **BS3** – The model can have Activities.
- **BS4** – The model can have Gateways.
- **BS5** – The model can have Connections.
- **BS6** – The model can have Swimlanes.
- **BS7** – The model can have Artefacts.
- **BS8** – Events are Start Events, End Events, or Intermediate Events.
- **BS9** – Events have a mode and a type.
- **BS10** – Activities are either Tasks or Processes.
- **BS11** – Gateways have a type.
- **BS12** – Connections are Sequence Flows, Message Flows, or Associations.
- **BS13** – Swimlanes are either a Pool or a Lane.
- **BS14** – Artefacts are Data Objects, Groups, or Annotations.
- **BS15** – Pools can contain multiple Lanes.

G.1.2 Behavioural requirements

Behaviour is expressed in model instances only, using the model elements to express behaviour. No behavioural requirements have been specified.

G.1.3 Constraints

- **BC1** – Connections must have exactly one source and one target.

G.2 Stock Management scenario requirements

G.2.1 Structural requirements

- **SS1** – A customer has a portfolio, and a standing which can be either good or bad.
- **SS2** – Shares are either common stock or preferred shares.
- **SS3** – Preferred shares may be convertible in which case they are convertible preferred shares which have a conversion rate and an earliest conversion date.
- **SS4** – An order is placed to either buy or sell a given number of stocks.
- **SS5** – A portfolio belongs to a customer and holds all stock he owns.
- **SS6** – A customer has a portfolio at the stock brokerage.
- **SS7** – A broker works for the stock brokerage. He has a speciality which can be either as a seller or buyer of stock.
- **SS8** – A broker handles orders from customers.
- **SS9** – A customer holds credentials which are used to identify with the stock brokerage.

G.2.2 Behavioural requirements

- **SB1** – A customer can check his portfolio.
- **SB2** – A customer can issue an order to buy or sell stock.
- **SB3** – A customer can review his portfolio.
- **SB4** – A customer can wait for his order to be fulfilled.

G.2.3 Constraints

- **SC1** – A customer cannot sell more shares than are in his portfolio.
- **SC2** – A customer in bad standing cannot buy preferred stock.
- **SC3** – A customer can only login with the correct credentials.
- **SC4** – A customer has at least one portfolio.
- **SC5** – Only a broker with the right specialisation can handle an order.
- **SC6** – A customer in bad standing cannot have more than two portfolios.

G.3 Tangible User Interface requirements

G.3.1 Structural requirements

- **TS1** – A widget has one or more handles.
- **TS2** – A handle can be bound or unbound.
- **TS3** – A handle has a continuous attribute, Domain Presence in the range]0;1].
- **TS4** – A widget holds an ID, a position, rotation angle, and clearable state.
- **TS5** – A widget may hold a visualisation.
- **TS6** – Widgets can be differentiated by their mode.
- **TS7** – A widget can have a state.

G.3.2 Behavioural requirements

- **TB1** – Dropping a handle on the table may bind the widget the handle is associated with to the underlying.
- **TB2** – Lifting the handle removes the Domain Presence of said handle.
- **TB3** – Shaking clears the mode of all widgets whose mode is clearable.
- **TB4** – Shaking removes the anchored entity from widgets whose mode is not clearable.

- **TB5** – A widget can be activated.
- **TB6** – A widget can be rotated.
- **TB7** – A widget can be dragged or slid over the canvas.
- **TB8** – Rotating a widget in the Zoom mode zooms the viewport in.
- **TB9** – Rotating a widget in the View mode rotates the viewport left.
- **TB10** – Rotating a widget in the Expander mode rotates the attached visualisation left.
- **TB11** – Moving two Spreader widgets apart increases the canvas space in between them.
- **TB12** – Moving two Spreader widgets together decreases the canvas space between them.
- **TB13** – Sliding a widget in the Sword mode across all of model entity deletes it.
- **TB14** – Sliding a widget in the Sword mode over parts of a model entity clears its contents.
- **TB15** – Moving a widget in Hand of God mode moves the grabbed component(s) as well.
- **TB16** – Activating a widget on the canvas depends on the widgets mode and the underlying canvas area. Consult Table G.1.
- **TB17** – Dropping the Stamp on the canvas creates the model entity the Stamp is currently imprinted with.

G.3.3 Constraints

- **TC1** – A widget has at least one handle.
- **TC2** – The rotation angle of a widget in the Spreader mode is modular to 45°.
- **TC3** – The enclosed space drawn from a Lasso must not cut across canvas sub-areas.

Canvas area	Widget mode	Desired Behaviour
Free space	Knob	Anchor the widget to enable the sliding of the viewport.
	Hand of God	Release the previously grabbed entity.
	Lasso	Draw an enclosed space creates a canvas sub-area.
	Chain	Show radial menu to select target component.
	Clone	Place copy of the component(s) grabbed by the first handle.
	Teleporter	Cut the component(s) grabbed by the first handle and place them at the second handles location.
Toolbox - Model entities	Stamp	Switch the stamp to the underlying model entity.
	Clone	Change the type of the model entity grabbed by the first handle in respect to the underlying toolbox model entity.
Toolbox - Modelling aids	Any	Attribute a new mode.
Model entity	Chain	Show radial menu to select connector or component.
	Hand of God	Grab the underlying model entity.
	Clone	Grab the underlying model entity.
	Teleporter	Grab the underlying model entity.
Any	View	Reset viewport to default.
	Expander	Hide respectively show sub-process.
	Wormhole	Anchors the wormhole end-point to the enclosing canvas area.

Table G.1: Table of behavioural requirements depending on activation zone

Appendix H

Ideation Scenario companion document

The following chapters will go through the process of creating the final model step by step. It is recommended that all users make themselves familiar with the table and the feel of the Sifteo cubes which are used as physical handles to the widgets.

Focus a *Lanes*

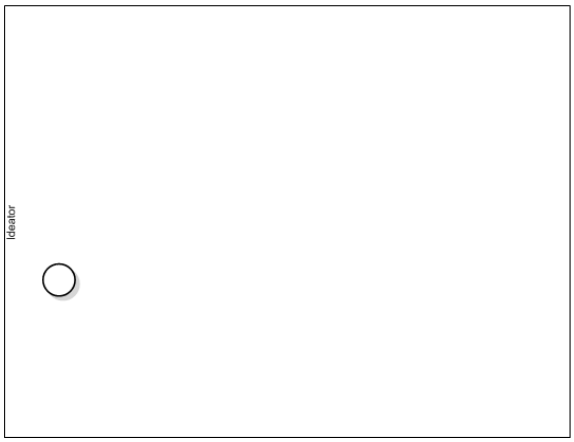
In order to facilitate modelling, lanes that are not in the focus of the modelling can be blended out.

Scenario

The user picks a *Zoom Widget* and places it on the lanes. He can then rotate the widget clockwise to increase the focus on the *Lanes* or activate it to immediately maximise the zoom factor.

Creating a *StartEvent*

Every process has to begin with a start event. These events are used to symbolise the instantiation of a process and, in turn, the associated tasks. Since the process of ideation is started by the ideator out of the blue, a global *StartEvent* is used. The *StartEvent* can however not just magically appear. To that end, the table displays a toolbox to the side of the modelling area. This toolbox is not unlike a colouring palette and contains all common modelling components.

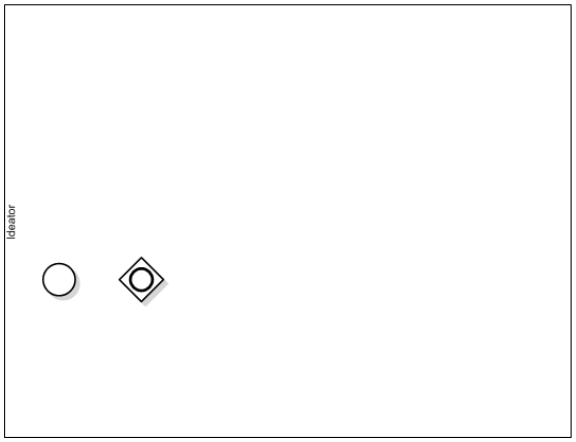


Scenario

The user picks a *Stamp Widget* and places it onto the *StartEvent*, a circle with a thin border, in the toolbox. He then activates the stamp which "imprints" the *StartEvent* onto the stamp. He then positions the stamp onto the space mirroring the placement of the *StartEvent* on the figure and activates the stamp again.

Creating a *Gateway*

A *Gateway* is used to split or link the flow of a process. Since the ideator can choose from different options, execute them and then select a different option, a *Gateway* is used as a splitting and reuniting point.

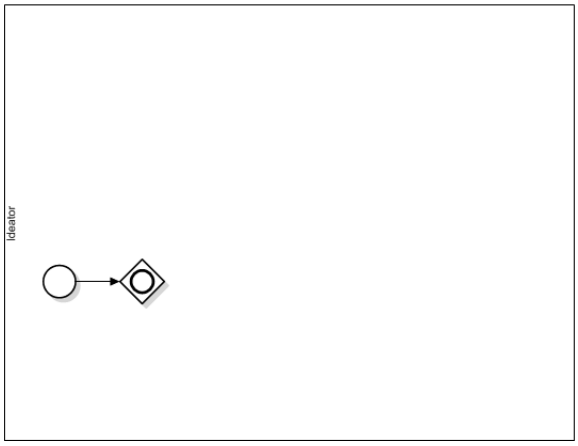


Scenario

The user picks a *Stamp Widget* and places onto the *Gateway*, a circle with a think border, in the toolbox. He then activates the stamp which "imprints" the *Gateway* onto the stamp. He then positions the stamp onto the space mirroring the placement of the *StartEvent* from the figure and activates the stamp again.

Link *StartEvent* and *Gateway*

In symbolise the link between components and the direction of the execution flow, BPMN2 uses flow connectors like arrows which can take different shapes. For this short scenario we will only use the plain arrow. In the first step we want to create a link between the two components we just created.



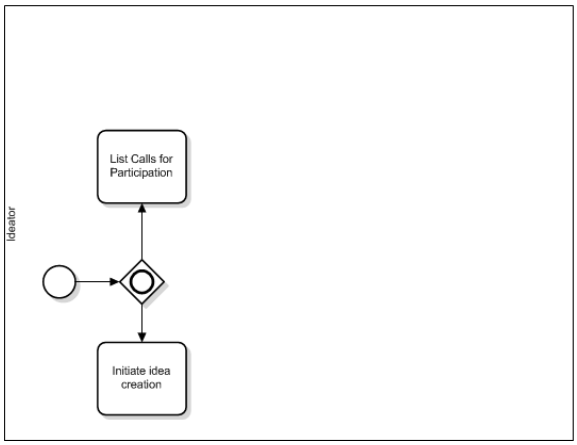
Scenario

The user takes two physical handles and places one on the tail of the arrow in the toolbox and the other on the head. He then activates either or both, linking them together as one widget. He proceeds to place the handle carrying the tail of the arrow on the *StartEvent* and the head on the *Gateway*. An arrow will be dynamically displayed to show how the arrow will be positioned. By activating the handle carrying the arrowhead, the arrow is finalised.

Creating the first batch of *Tasks*

Tasks are conveying the actions in BPMN2. They deliver the main messages and take many nuances. It is therefore imperative that the right

kind of tasks be chosen. In this case, the ideator chooses two plain tasks but applies them differently.



Scenario

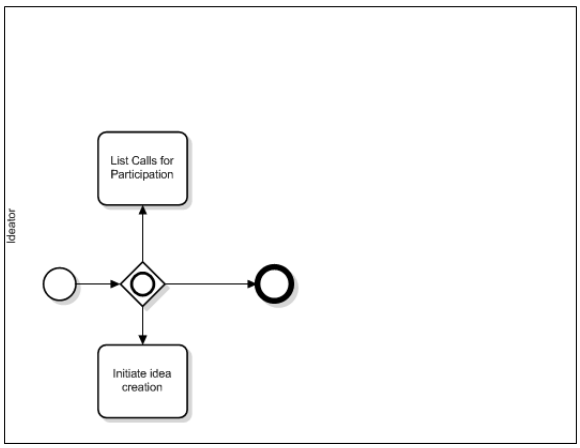
The ideator uses the *Stamp Widget* . He places it over the *Tasks* icon in the toolbox and activates the *Stamp Widget*. From the radial menu he chooses the plain *Tasks* and activates the *Stamp Widget* a second time. He then proceeds to apply the *Tasks* to the canvas as shown in the figure. He places only **one** *Tasks*. The ideator proceeds to link the *Gateway* and *Tasks* by the same scenario as illustrated in H.

Scenario

From the previous linking of *Gateway* and *Tasks*, the ideator still has two handles to apply arrows. He places the tail on the *Gateway* and the head onto an empty space, where in the figure the arrow tot he second *Tasks* would end. He then actives the widget. A radial menu will guide the ideator through the selection of end point component for the arrow. He will first select "*Tasks*" and then the plain "*Tasks*" from the second level.

All things must end in an event

An *EndEvent* is the logical counterpart of a *StartEvent*. It is used to end all flows and indicate that the desired business interaction has come to a close.

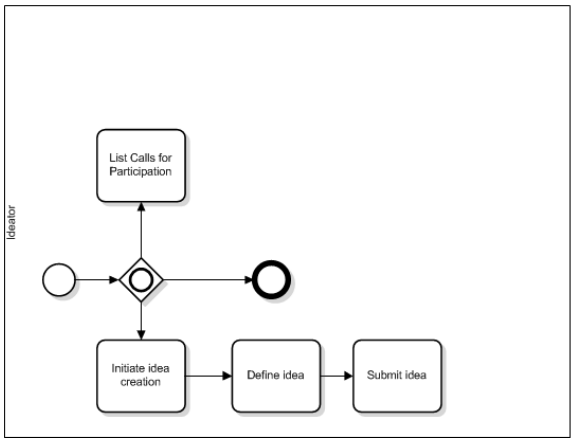


Scenario

The ideator uses the *Stamp Widget* to first associate the *EndEvent* with the widget using the toolbox and than apply it onto the canvas. He will then link the *Gateway* and *EndEvent* using the linking method described in H.

Chaining components

During the modelling of a system, some tasks are very clear and well known. These "strings" of tasks usually require redundant modelling actions. A specialised widget, the *Chain Widget* is used to make this process easy to apply in TUIs.

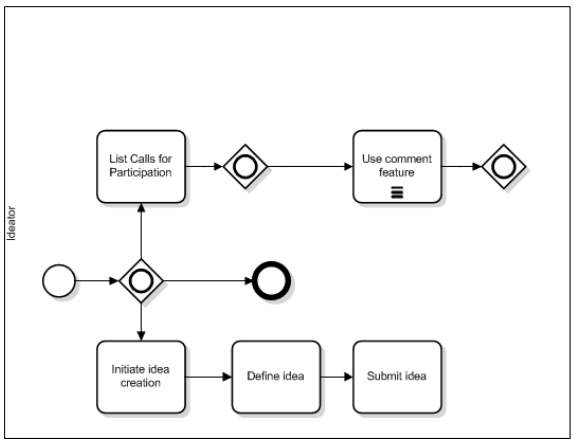


Scenario

The ideator places the *Chain Widget* onto the *Tasks* he wants to start the chain at. In this case it would be the lower left *Tasks* as can be seen from the figure. The ideator activates the widget which will display all available flow connectors in a radial menu. The ideator selects the appropriate connector, the arrow in our case and activates the widget a second time. He can no proceed to slide the widget to the left, drawing the arrow along with the handle as it it were attached. once at a suitable spot on the canvas he can activate the widget once more, showing a radial menu of suitable end point components. The ideator should first select "*Tasks*" and then the plain "*Tasks*" from the second level. This process is repeated once more to create the second first select "*Tasks*" and then the plain "*Tasks*" from the second level. The process is repeated once more to create the second *Tasks*.

Chaining mixed components

Chaining can not only be used with similar items but also across multiple type of components.

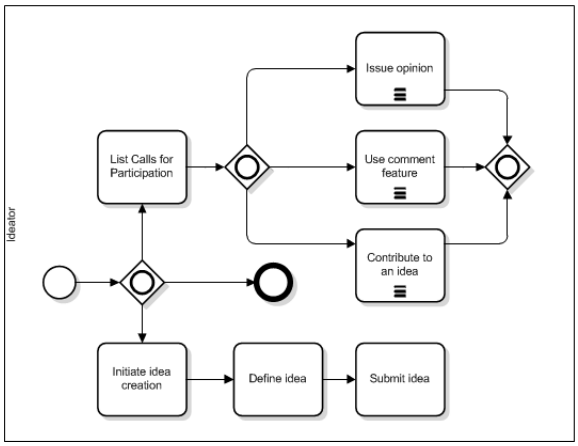


Scenario

Analogous to H. From the radial menus the ideator can choose the components matching the figure.

Chaining or Stamping

The ideator should create and link two more *Tasks*. It is up to him how he wants to create those *Tasks*.



Alternative 1

The ideator uses a *Stamp Widget* and assigns it to the parallel *Tasks* by first activating it on the general *Tasks* and then, from the radial menu, selecting the parallel *Tasks*. He then creates those *Tass* on the canvas as indicated by the figure. Subsequently, he attributes a second handle to the widget and associates the arrow flow connector to the *Stamp Widget* in order to create the four needed links.

Alternative 2

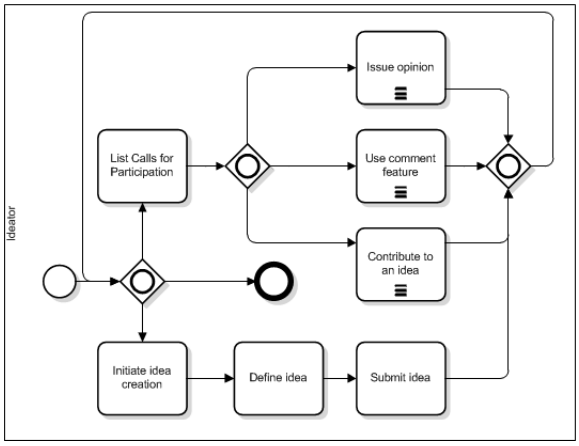
The ideator uses a *Chain Widget*. He places it on the *Gateway* and activates it. Choosing the flow connector and a direction to move in, he proceeds to create a parallel *Tasks* from the second level radial menu. From that parallel *Tasks* he creates the missing link. In exactly the same way he produces the mirrored second *Tasks*.

Alternative 3

The ideator can mix the use of the *Stamp Widget* and *Chain Widget*.

Final linking

The ideator finalises the *Process*, adding the last links to the model.



The ideator uses a *Chain Widget* and places it on the rightmost *Gateway*. After activating the widget he chooses the arrow flow connector and loops it back into the leftmost *Gateway*, completing the process loop. He now creates the last missing link from the last of the three *Tasks* at the bottom to the rightmost *Gateway* using the same method.

Appendix I

Weighting evaluation criteria

To find a Visual Modelling Language (VML), such as UML, suitable to express the modelling of Business Process Model and Notation 2 (BPMN2) models using a Tangible User Interface (TUI), several criteria have been distilled. It is important to weigh these criteria carefully and gather as much feedback as possible. Therefore, this small study has been devised to gauge the user perspective.

The following six broad criteria are used to judge the suitability of VML. Please rate how important it is for you that a VML you would want to use fulfils the criterion?

	Not important at all	Somewhat important	Neutral	Important	Very important
Tool support					
Semantics & Transformation					
Expressivity					
Usability					
Error checking					
Verification					

Appendix J

Widget list

Table J.1: A listing of all widgets conceived in cooperation with Bicheler.

Name	Physical Action	Intent	Result
Zoom View	The actor places the widget on the surface and rotates the widget clockwise respectively counter-clockwise.	The actor wants to increase or decrease the zoom factor of the canvas with respect to the handles current position.	The virtual camera zooms and is repositioned based on the current handle position.
Rotate View	The actor places the widget on the surface and rotates the widget clockwise resp. counter-clockwise.	The actor wants to rotate the canvas.	The canvas rotates in the direction the widget is rotated.
Drag Canvas	The actor places the widget on the surface and drags it in the desired direction.	The actor wants to reposition the canvas, using the widget as a virtual knob.	The canvas is repositioned, opposite to the widget movement direction, producing the drag effect.

Continued on next page

Table J.1 – *Continued from previous page*

Name	Physical Action	Intent	Result
Stamp	The actor moves the widget onto an element of the stamp toolbox, assigns the element by activating the widget and then stamps one or multiple elements onto the canvas.	The actor wants to select an element from the stamp toolbox and place one or multiple instances of it on the canvas.	The desired number of elements are placed on the canvas.
Link	The actor chooses the type of link the same way as the stamp but using the link toolbox. Then he moves the two handles of the link widget over two zones on the canvas and after seeing a preview of the link, fixes it by activating either handle.	The actor wants to link existing zones using a specific kind of link., Using the preview, the actor can establish a temporary link by leaving the handles on the respective zones.	The existing zones are linked by the new type of link.
Chain	The actor slides the widget from a zone to a destination and activates the handle to confirm and select the kind of element to be created.	The actor wants to create a new element in a process by creating a link and element from an existing component to the newly created one.	A new component with link to it is created.
Wormhole [Create]	The actor places two handles to create a wormhole between them.	The actor wants to link two different locations on the canvas.	Both locations are linked and accept elements to be transferred through it.
Wormhole [Transfer]	The actor swipes zones into the vicinity of one end of the wormhole to transfer them to the other.	The actor wants to transfer model elements from one end to the other.	All model elements pushed through one end will appear on the other end.

Continued on next page

Table J.1 – *Continued from previous page*

Name	Physical Action	Intent	Result
Spreader	The actor places two handles on the surface and either brings them closer together or increases the distance between them while they touch the surface.	The actor wants to increase or decrease the space available between or within existing zones.	Space pours in or is removed from the area between the horizontal or vertical axes implied by the widget placement. Connectors are shortened resp. lengthened.
Sword	The actor "cuts" across the canvas in a slashing motion, deleting all zones the handle touches.	The actor wants to remove or delete components from the canvas.	The "cut" zones are deleted from the canvas.
Hand of God	The actor places the handle on an existing zone and activates it to "grab" it. The actor moves the handle to its destination and activates the widget again to "drop" the held zone.	The actor wants to move a component around on the canvas or drags the edge of a zone to resize the contained elements.	The element is moved from its initial position to the destination. When the handles is moved on the surface, a preview is shown.
Lasso	The actor places the widget on the surface and activates it. He then draws a closed shape around the desired zones and activates the widget once more.	The actor wants to select a subsection containing certain elements of the canvas.	The space enclosed in the shape is marked and treated as separate area of the canvas until the widget selection is cancelled.