UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Sören Blom

J^3D_L: Formalizing a Visual Language for Java Class Relations by Means of Relation Algebra

Diplomarbeit

April 22, 2005

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl für Software-Technologie

Gutachter:
Prof. Dr. E.-E. Doberkat
Dr. Alexander Fronk

GERMANY · D-44221 DORTMUND

To my parents

Acknowledgments

"That's quite an undertaking you picked yourself there", was the dry comment of Professor Doberkat regarding my plans for my Masters' Thesis. In the following weeks and months, I often thought back on these words. While my studies so far have been mostly fun and sometimes boring, I had not expected the range of emotions that my first attempt at research would have provided for me. The identification with "my" formulas created frustration but even more satisfaction when after hours of trying everything worked out. Needless to say that I learned a lot about scientific working in general and relation algebra in particular.

Many thanks go to Alexander Fronk for patience and pressure that he applied in the right balance, his ever-ready support and countless valuable remarks on my thesis.

Many thanks go to Jennifer who ensured that I survived the writing of the thesis mentally and bodily, her tolerance for a vacantly and tired husband that spent many weekends at the desk and her proof-reading that ensured my English is understandable.

Most of all I want to thank my parents for the ongoing support for many years that made me reach this point.

Final thanks go to Remy and Modibo who, if for nothing else, were good for comic relief.

Sören Blom, April 2005

Contents

List of Figures

Part I

Introduction

Chapter 1

Introduction

Nowadays, any student of computer science or practitioner of software engineering will probably be knowledgeable of at least two things: an object-oriented programming language, and corresponding with that, a visual notation for designing object-oriented software. Examples of the first are Java, C++ or C#, while examples for the latter are most prominently the Unified Modelling Language (UML) [OMG03], or its predecessor the Object Modelling Technique (OMT) [RBP+91].

The visual notation allows people to talk about properties of the program without referring to the source code, which is particularly interesting for object-oriented software with its focus on structure. The interesting aspects of a software are spread over several artefacts (files and folders representing classes and packages) but need to be viewed together with the relations between them also visible. The famous book on Design Patterns [GHJV94] depends on the visual notation (OMT) to define the patterns, next to the textual description.

UML is a two-dimensional visualization of software, but the human cognitive system evolved inside a three dimensional world, and so the idea is near to apply the third dimensions to the visualization of software, in this case the structural aspects of object-oriented software. When lifting the visualization up to three dimension one is not limited to merely replacing the 2D shapes to their 3D equivalences. A variety of visualization techniques exist which make particular use of the third dimension to achieve a better presentation of the information or to ease understanding.

The creation of three-dimensional models must be, of course, supported by editing environments, which ideally give feedback to the user regarding the validity of the model. This way an editor turns from a mere "drawing tool" into a tool for visual programming. The visualization techniques come with their own spatial requirements which should also be part of the feedback that the editor gives the user during editing a model.

In order to determine whether or not a 3D scene inside an editor is valid,

the availability of a visual language is helpful. Instead of having a number of validity checks being hard coded directly into the editors drawing functions, the visual language can be understood as an implementation-independent description of valid models; the same way programming languages are independent from the hardware design their programs are executed on.

The goal of this thesis is the definition of J^3D_L, a visual language for the three-dimensional modeling of Java class structures. As the title of the thesis suggests, we use relation algebra to define the language.

Vise3D

This thesis is part of a larger research effort, the VISE3D project at the Software Technology Group, University of Dortmund. The website describes the project's goals as follows:

> *"[...] the aim of the project is to introduce 3D visualization in software engineering tasks where appropriate. Possible application areas range from employing 3D instead of 2D diagrams for modeling purposes such as UML diagrams, to the field of visualizing management information such as version management."* [V3D]

We will give a short overview of the project's milestones, to then explain how the work of this thesis fits in. When began in 1999, the first major step was to create an 3D editor as part of a master's thesis [Eng00], which allowed the user to browse Java code in 3D. In this thesis and in [AF00, AF02], a graphical notation was developed together with selecting visualization techniques that arrange the graphical elements in certain ways, based on their meanings. In 2003, Berghammer & Fronk published a paper [BF03] which described the idea of specifying a visual language for modeling object-oriented structures three-dimensionally by relation-algebraic constraints. Their paper is part of a more general interest to explore the application of relation-algebraic methods in software engineering, such as searching for design patterns [BF04]. The tool they proposed for evaluating relation-algebraic expressions is RELVIEW [BS97, RHP05].

In order to implement tools that allow not only the display but also the manipulation of 3D scenes representing object oriented structures, the EFFECTS framework was developed as part of a year-long student project [DFSH04], which supports the creation of 3D editors as ECLIPSE [Ecl] plug-ins. Independently from EFFECTS, another editor was implemented as part of a master's thesis [Roh04], to allow better possibilities for the evaluation of the benefits of 3D visualization techniques for object-oriented code.

Yet another master's thesis [Tch04] took the first step of implementing an 3D editor for the manipulation of object-oriented software structures that

uses relation algebra to check a visual language phrased in constraints. The visual language in this thesis merely applied the few constraints developed in [BF03], as its main focus was to examine the challenges of implementation, not to define the language itself. It succeeded in being a proof of concept for the combination of 3D editing and relation algebra for visual languages.

This thesis will take the notation and visualization techniques to develop a language declaratively using relation-algebraic constraints as described by [BF03], so it can be the basis for editors such as [DFSH04, Roh04] while considering the experiences made in [Tch04].

Layout of the Thesis

The chapter of the thesis are grouped into four parts. Part I includes, next to this chapter, the introduction of important preliminaries (Chapter 2), and a description of the approach and goals of this thesis (Chapter 3). In Part II we define the different parts of the language, its basics in Chapter 4 and the different visualization techniques in Chapters 5-7. After that we show in Part III how the different parts of the language can be integrated. Finally Part IV presents thoughts that go beyond the actual language definition. Chapter 11 presents consequences of the language design for the implementation, Chapter 12 sums up the thesis and Chapter 13 discuss further directions that research in this area could take.

Chapter 2

Preliminaries

In the last section, we described the goal of this thesis: to define a three-dimensional visual language for Java class structures using relation algebra. Before moving towards that goal, we want to provide background information about the mathematical and technical concepts and tools that we use.

Sect. 2.1 gives an introduction into relational algebra and some relational properties we will use frequently in the later chapters. The next two sections deal with the question of how the relations can be used to define a language. Sect. 2.2 introduces the idea of different diagram layers with which a 3D scene can be described with relations, and Sect. 2.3 explains how a visual language can be described declaratively using constraints. In Sect. 2.4 we give a short introduction to the visualization techniques that are used in $\mathrm{J^3D_L}$.

2.1 Relational Algebra

The everyday notion of a (binary) *relation* is straightforward. It is a set of pairs; more precisely, a subset of the Cartesian product $X \times Y$ of two sets X and Y. The notion of *relatedness* has highly descriptive powers when applied to real-world problems, and relations are in wide-spread use in the sphere of computer science (e.g. relational database systems).

For the purpose of this thesis (and many other papers employing relational concepts in computer science) we need a more formal calculus, *relation algebra* (aka relational algebra). This takes relations and augments them with algebraic concepts, thus allowing to perform calculations with the relational expressions. The relational calculus was first examined in the middle of the 19th century by A. De Morgan, C.S. Pierce and E. Schröder. In the 1940's it was developed further by A. Tarski (for a comprehensive historical description of relation algebra see [Mad91]).

In this section we give an introduction to relation algebra and necessary preliminary knowledge. First we recapitulate the relational calculus, then

we combine this with the definition of relation algebra. Finally, we discuss how this will be applied in subsequent chapters. The whole section closely follows [SS93] and [BKS97, Chapter 1], and to a lesser degree [Ber03] and [Beh98]. Other subjects such as set theory or propositional and predicate logic are assumed to be known.

2.1.1 Relational Calculus

A (binary) *relation* R is a set of pairs, and as such, a subset $\mathsf{R} \subseteq X \times Y$ of the Cartesian product of X and Y. In the case of $X = Y$ the relation is called *homogeneous* and *heterogeneous* otherwise. Instead of $R \subseteq X \times Y$ we write $\mathsf{R} : X \leftrightarrow Y$ and $\mathsf{R}_{a,b}$ whenever $(a, b) \in \mathsf{R}$ holds. $[X \leftrightarrow Y]$ is the set of all relations between X and Y, called the *type* of the relation. Some important relations include:

$\mathbf{L} : \boldsymbol{X} \leftrightarrow \boldsymbol{Y}$ the *universal* relation, which contains all pairs out of $X \times Y$.

$\mathbf{O} : \boldsymbol{X} \leftrightarrow \boldsymbol{Y}$ the *empty* relation, which contains no pair out of $X \times Y$.

$\mathbf{I} : \boldsymbol{X} \leftrightarrow \boldsymbol{X}$ the *identity* relation, defined on homogeneous relations, contains only identical pairs $\mathbf{I} := \{(x, x) : x \in X\}$.

For all three of the above relations, we usually do not specify the type of the relation when it can be deducted from the context and treat $\mathbf{L}$, $\mathbf{O}$ and $\mathbf{I}$ as a family of relations respectively. For the remainder of this section we will assume that all example relations are of the type $[X \leftrightarrow Y]$ unless otherwise noted.

The following operations exists on relations:

union: for two relations, the union includes all pairs that are in the one or the other relation. $(\mathsf{R} \cup \mathsf{S})_{x,y} := \mathsf{R}_{x,y} \vee \mathsf{S}_{x,y}$.

intersection: for two relations, the intersection includes all pairs that are part of both relations. $(\mathsf{R} \cap \mathsf{S})_{x,y} := \mathsf{R}_{x,y} \wedge \mathsf{S}_{x,y}$.

complement: for a given relation, the complement is the relation which includes all pairs that are not part of a relation (relatively to $X \times Y$). $\mathsf{R}^{\mathsf{T}} := (X \times Y) \setminus R$.

converse: is the "rotation" (aka "transposition") of a given relation $\mathsf{R}^{\mathsf{T}} := \{(x, y) : \mathsf{R}_{y,x}\}$. This operation changes the type of the resulting relation.

composition: combines two relations $\mathsf{R} ; \mathsf{S} := \{\exists z : \mathsf{R}_{x,z} \wedge \mathsf{S}_{z,y}\}$. Note that this requires matching signatures of the relations, so that $\mathsf{R} : X \leftrightarrow Z$ and $\mathsf{S} : Z \leftrightarrow Y$ become $(\mathsf{R} ; \mathsf{S}) : X \leftrightarrow Y$. The composition is treated as the *multiplication* of relations. This motivates the notation of R^i, recursively defined as $\mathsf{R}^i := \mathsf{R} ; \mathsf{R}^{i-1}$ and $\mathsf{R}^1 := \mathsf{R}$, $\mathsf{R}^0 := \mathbf{I}$.

With these operations we can now introduce two important concepts:

domain: contains all elements of X that appear in a pair of R. $\mathsf{dom}(\mathsf{R}) := \{x : \exists y\ \mathsf{R}_{x,y}\} = \mathsf{R}\,;\mathbf{L}$.

range: contains all elements of Y that appear in a pair of R. $\mathsf{ran}(\mathsf{R}) := \{y : \exists x\ \mathsf{R}_{x,y}\} = \mathsf{R}^\mathsf{T}\,;\mathbf{L}$.

Having the operations at hand, we can now define some properties that will be of interest later on. Let R be a homogeneous relation. It is called:

reflexive $:\Leftrightarrow \mathbf{I} \subseteq \mathsf{R}$

irreflexive $:\Leftrightarrow \mathsf{R} \subseteq \overline{\mathbf{I}}$

transitive $:\Leftrightarrow \mathsf{R}\,;\mathsf{R} \subseteq \mathsf{R}$

symmetrical $:\Leftrightarrow \mathsf{R} \subseteq \mathsf{R}^\mathsf{T}$

asymmetrical $:\Leftrightarrow \mathsf{R}^\mathsf{T} \subseteq \overline{\mathsf{R}}$.

A relation that is irreflexive and transitive (and therefore asymmetric) is a *strict partial order*. In the case of a heterogeneous relation we call R:

univalent $:\Leftrightarrow \mathsf{R}^\mathsf{T}\,;\mathsf{R} \subseteq \mathbf{I}$

total $:\Leftrightarrow \mathsf{R}^\mathsf{T}\,;\mathbf{L} = \mathbf{L}$

injective $:\Leftrightarrow \mathsf{R}\,;\mathsf{R}^\mathsf{T} \subseteq \mathbf{I}$

surjective $:\Leftrightarrow \mathbf{L}\,;\mathsf{R} = \mathbf{L}$

A relation that is univalent and total is called a *mapping*, and we write $\mathsf{R}(x)$ to refer to the element y with $\mathsf{R}_{x,y}$. As R is total y must exist; and as R is univalent it must be unique.

2.1.2 Abstract Relation Algebra

Relations are normally thought of as *sets*, with pairs of elements that constitute the elements of the relations. The operators of relations are defined by making reference to the elements of the relations as we did above. The *abstract* relation algebra is an axiomatic definition of relation algebra that is not limited to relations being sets. We want to give a short overview of how the abstract relation algebra is defined. First, we need to introduce two important theorems:

Theorem 2.1 (Schröder Equivalences). *For three relations* Q, R *and* S *(assuming matching signatures) holds:*

$$\mathsf{Q}\,;\mathsf{R} \subseteq \mathsf{S} \Leftrightarrow \overline{\mathsf{S}}\,;\mathsf{R}^\mathsf{T} \subseteq \overline{\mathsf{Q}} \Leftrightarrow \mathsf{Q}^\mathsf{T}\,;\overline{\mathsf{S}} \subseteq \overline{\mathsf{R}}.$$

□

Theorem 2.2 (Tarski Rule). *For all relations* R *holds:*

$$\mathsf{R} \neq \mathbf{O} \Leftrightarrow \mathbf{L} \,;\mathsf{R}\,;\mathbf{L}.$$

□

Now we can give a compact definition of what an (abstract) relation algebra is:

Definition 2.1 (Abstract Relation Algebra). *An abstract relation algebra is a 9-tuple* $(A, \sqcup, \sqcap, \bar{\cdot}, \bot, \top, \,;, ^{\mathsf{T}}, \mathsf{I})$ *with:*

- $(A, \sqcup, \sqcap, \bar{\cdot}, \bot, \top)$ *being a complete, atomic Boolean algebra,*
- $\boxed{;}$ *(the composition operator) being an associative mapping and* I *as its left and right identity,*
- $\boxed{^{\mathsf{T}}}$ *(the converse operator) being a mapping,*
- *the Schröder equivalences and the Tarski rule hold.*

We can obtain a *concrete* relation algebra by fixing a relation algebra $([X \leftrightarrow Y], \cup, \cap, \bar{\cdot}, \emptyset, \,;, ^{\mathsf{T}}, \mathsf{I}, \mathsf{L})$. Whenever we talk about relations in subsequent chapters, it is implied that they are part of such a concrete relation algebra. The equalities and inequalities shown in Table 2.1 must hold for $\boxed{;}$ and $\boxed{^{\mathsf{T}}}$. For reference in later chapters we will label them.
This selection of laws shown in Table 2.1 is called the *arithmetic* of relation algebra and we will make intensive use of it when dealing with algebraic expressions. We are now turning towards some constructs that enable us to use special situations such as subsets, relations between more than two elements, and closures.

2.1.3 Representing Sets and Elements

When working inside relation algebra we need a way to express set and subset memberships using relations and the special case of single elements (sets with only one element). The common way of doing this is by using relations with special properties, known as *vectors*.

Vectors

A (concrete) relation can be given as a matrix with all elements of X as rows, those of Y as columns and the cells containing either "1" or "0" depending on if the two elements are related or not. One might run into a relation v where all rows of this matrix are either constantly "1" or constantly "0", that is an element of X is either related to all elements of Y or no element. This property is equivalent to $\mathsf{v}\,;\mathbf{L} = \mathbf{L}$. In this case the matrix can be reduced to only one column.

Table 2.1: The arithmetic of (abstract) relation algebra.

Reference	Expression
A1	$\mathsf{R};\mathbf{O}=\mathbf{O};R=\mathbf{O}$
A2	$\mathsf{R}\subseteq\mathsf{S}\Rightarrow\mathsf{Q};\mathsf{R}\subseteq\mathsf{Q};\mathsf{S}$ and $\mathsf{R}\subseteq\mathsf{S}\Rightarrow\mathsf{R};\mathsf{Q}\subseteq\mathsf{S};\mathsf{Q}$
A3	$\mathsf{Q};(\mathsf{R}\cap\mathsf{S})\subseteq\mathsf{Q};\mathsf{R}\cap\mathsf{Q};\mathsf{S}$ and $(\mathsf{R}\cap\mathsf{S});\mathsf{Q}\subseteq\mathsf{R};\mathsf{Q}\cap\mathsf{S};\mathsf{Q}$
A4	$\mathsf{Q};(\mathsf{R}\cup\mathsf{S})=\mathsf{Q};\mathsf{R}\cup\mathsf{Q};\mathsf{S}$ and $(\mathsf{R}\cup\mathsf{S});\mathsf{Q}=\mathsf{R};\mathsf{Q}\cup\mathsf{S};\mathsf{Q}$
A5	$\mathsf{R}^{\mathsf{T}^{\mathsf{T}}}=\mathsf{R}$
A6	$(\mathsf{R};\mathsf{S})^{\mathsf{T}}=\mathsf{S}^{\mathsf{T}};\mathsf{R}^{\mathsf{T}}$
A7	$\overline{\mathsf{R}^{\mathsf{T}}}=\overline{\mathsf{R}}^{\mathsf{T}}$
A8	$(\mathsf{R}\cup\mathsf{S})^{\mathsf{T}}=\mathsf{R}^{\mathsf{T}}\cup\mathsf{S}^{\mathsf{T}}$
A9	$(\mathsf{R}\cap\mathsf{S})^{\mathsf{T}}=\mathsf{R}^{\mathsf{T}}\cap\mathsf{S}^{\mathsf{T}}$
A10	$\mathsf{R}\subseteq\mathsf{S}\Rightarrow\mathsf{R}^{\mathsf{T}}\subseteq\mathsf{S}^{\mathsf{T}}$

Definition 2.2 (Vector). *A* vector *is a relation of the type* $[M\leftrightarrow\mathbf{1}]$. *The* $\mathbf{1}$ *is a set with exactly one arbitrary but fixed element* $\blacklozenge$.

The following arithmetical rules are valid for vectors:

1. if v is a vector, then also $\overline{\mathsf{v}}$ and $\mathsf{R};\mathsf{v}$, are vectors, with R being any given relation.
2. if v and w are vectors, then also $\mathsf{v}\cup\mathsf{w}$ and $\mathsf{v}\cap\mathsf{w}$ are vectors.
3. the vectors constitute a sublattice of the relations.

It is easy to see how a vectors model (sub)sets: vector $\mathsf{v}:M\leftrightarrow\mathbf{1}$: represents a subset $\{x:\mathsf{v}_{x,\blacklozenge}\}$ of the base set M. As the vectors are a sublattice of relations, all set-theoretic operations on a vector result in another vector.

Points

Sometimes it is necessary to model individual elements. Single elements can be understood as sets with exactly one element and can therefore be modeled as special cases of vectors.

Definition 2.3 (Point). *A point is a vector with exactly one entry.*

In the lattice of vectors, points are the atoms. For point p the following arithmetical rules hold, additionally to those for vectors:

- $\mathsf{p}\,;\mathsf{p}^{\mathsf{T}} \subseteq \mathsf{I}$,
- $\mathsf{L}\,;\mathsf{p} = \mathsf{L}$.

2.1.4 Residuals

The residuals are usually introduced as a concept to model solving linear (in)equations. We look at them purely as a tool to translate certain predicate logical expressions.

right residual: the right residual is the largest solution X of $\mathsf{R}\,;\mathsf{X} \subseteq \mathsf{S}$. Using the operations from above it can be written as $\mathsf{R}\backslash\mathsf{S} = \overline{\mathsf{R}^{\mathsf{T}}\,;\overline{\mathsf{S}}}$.

left residual: the left residual is the largest solution X of $\mathsf{X}\,;\mathsf{R} \subseteq \mathsf{S}$. Using the operations from above it can be written as $\mathsf{R}/\mathsf{S} = \overline{\overline{\mathsf{S}}\,;\mathsf{R}^{\mathsf{T}}}$.

symmetrical quotient: the symmetrical quotient is both left and right residual $\mathsf{syq}(\mathsf{R},\mathsf{S}) :\Leftrightarrow \mathsf{R}\backslash\mathsf{S} \cap \mathsf{R}/\mathsf{S} = \overline{\mathsf{R}^{\mathsf{T}}\,;\overline{\mathsf{S}}} \cap \overline{\overline{\mathsf{S}}\,;\mathsf{R}^{\mathsf{T}}}$.

In the case of a concrete relation algebra it can be shown that:

1. $(\mathsf{R}\backslash\mathsf{S})_{b,c} \Leftrightarrow \forall a\ \mathsf{R}_{a,c} \rightarrow \mathsf{S}_{b,c}$
2. $(\mathsf{R}/\mathsf{S})_{b,c} \Leftrightarrow \forall a\ \mathsf{R}_{c,a} \rightarrow \mathsf{S}_{b,a}$
3. $\mathsf{syq}(\mathsf{R},\mathsf{S})_{b,c} \Leftrightarrow \forall a\ \mathsf{R}_{a,b} \leftrightarrow \mathsf{S}_{a,c}$.

2.1.5 Direct Product

Often there are situations when binary relations are not containing atomic elements, but the elements are tuples themselves. As an example, consider a relation $\mathsf{distance}_{<p_1,p_2>,x}$ which relates the distance x to the two points p_1 and p_2. To deal with this situation, we need measures to access the components of $< p_1, p_2 >$. The *natural projections* $\pi : M \times N \leftrightarrow M$ and $\varrho : M \times N \leftrightarrow N$ are defined as:

$$\pi_{<a,b>,a'} :\Leftrightarrow a' = a,\ \forall a, a' \in M \text{ and } \forall b \in N,$$
$$\varrho_{<a,b>,b'} :\Leftrightarrow b' = b,\ \forall a \in M \text{ and } \forall b, b' \in N.$$

The tuple (π, ϱ) is called the *direct product* of $M \times N$. In the abstract relation algebra the direct product is defined by four laws:

$$\pi^{\mathsf{T}}\,;\pi = \mathsf{I}, \qquad \varrho^{\mathsf{T}}\,;\varrho = \mathsf{I},$$
$$\pi\,;\pi^{\mathsf{T}} \cap \varrho\,;\varrho^{\mathsf{T}} = \mathsf{I}, \qquad \pi^{\mathsf{T}}\,;\varrho = \mathsf{L}.$$

The four laws imply that both projections are surjective functions and that for every element of M and every element of N a pair exists in $M \times N$. It can be shown that this characterization of the direct product is unique.

2.1.6 Closures

We introduce the *transitive closure* and *reflexive transitive closure* as they are known for graphs. We will use them to express certain "neighborhood" relationships of elements. If R is a homogeneous relation we call

$$\mathsf{R}^+ := \bigcup_{i \geq 1} \mathsf{R}^i = \bigcap \{H | \mathsf{R} \subseteq H, H \text{ transitive}\}$$

the *transitive closure*, and

$$\mathsf{R}^* := \bigcup_{i \geq 0} \mathsf{R}^i = \bigcap \{H | \mathsf{R} \subseteq H, H \text{ transitive, reflexive}\}$$

the *transitive reflexive closure*. They are the smallest transitive, respectively transitive and reflexive relations, to include R. For the two closures it holds that:

$$\mathsf{R}^+ = \mathsf{R}\,;\mathsf{R}^* \text{ and } \mathsf{R}^* = \mathsf{I} \cup \mathsf{R}^+.$$

2.1.7 Overview of Transformations and Equivalences

The objective in the subsequent chapters will be to transform properties of the language into purely relation-algebraic expressions, which are expressions that do not refer to individual elements of relations. This is considered advantageous in that component-free expressions are supposedly more compact, easier to comprehend, and less error-prone.

From Predicate Logic to Relation Algebra

Table 2.2 on the following page collects the previously defined properties and lists them as predicate logical as well as relation-algebraic expressions. We also give a reference number which we will use later on to refer to the exact transformation shown in this section.

Miscellaneous

If R is a mapping (univalent and total) we get

$$T13: \quad \mathsf{R}(x) :\Leftrightarrow \exists y\, \mathsf{R}_{x,y},$$

as for each x (total) must exist exactly one y (univalent).

A very helpful "trick" when transforming predicate logical statements into relation-algebraic ones is the "extension" of a statement with L using an appropriate signature. As *all* pairs of a given type are part of the universal relation, it is always true that one *specific* pair is part of it. Therefore the truth value of the predicate logic expression is not altered if we add such a statement about the universal relation. This is mostly helpful when trying

Table 2.2: Predicate logic expressions and their relation-algebraic transformations.

Reference	Predicate logic	Relational algebra
T1	$(x, y) \in \mathsf{R}$	$\mathsf{R}_{x,y}$
T2	$(x, y) \in \mathsf{R}$	$\mathsf{R}^{\mathsf{T}}_{y,x}$
T3	$x = y$	$\mathbf{I}_{x,y}$
T4	$\mathsf{R}_{x,y} \wedge \mathsf{S}_{x,y}$	$(\mathsf{R} \cap \mathsf{S})_{x,y}$
T5	$\mathsf{R}_{x,y} \vee \mathsf{S}_{x,y}$	$(\mathsf{R} \cup \mathsf{S})_{x,y}$
T6	$\neg\mathsf{R}_{x,y}$	$\overline{\mathsf{R}}_{x,y}$
T7	$\exists z\, \mathsf{R}_{x,z} \wedge \mathsf{S}_{z,y}$	$\mathsf{R}\,;\mathsf{S}_{x,y}$
T8	$\forall y\, \mathsf{R}_{x,y} \rightarrow \mathsf{S}_{x',y}$	$(\mathsf{R}/\mathsf{S})_{x,x'}$
T9	$\forall x\, \mathsf{R}_{x,y} \rightarrow \mathsf{S}_{x,y'}$	$(\mathsf{R}\backslash\mathsf{S})_{y,y'}$
T10	$\forall x\, \mathsf{R}_{x,y} \leftrightarrow \mathsf{S}_{x,y'}$	$\mathsf{syq}(\mathsf{R},\mathsf{S})_{y,y'}$
T11	$\forall x, y\, \mathsf{R}_{x,y} \rightarrow \mathsf{S}_{x,y}$	$\mathsf{R} \subseteq \mathsf{S}$
T12	$\forall x, y\, \mathsf{R}_{x,y} \leftrightarrow \mathsf{S}_{x,y}$	$\mathsf{R} = \mathsf{S}$

to compose two expressions, but their signatures do not match, especially when one relation is a vector. Take a look at the following example:

$$
\begin{aligned}
&\mathsf{R}_a \wedge \mathsf{S}_{a,b} \\
&\Leftrightarrow \mathsf{R}_{a,\blacklozenge} \wedge \mathbf{L}_{\blacklozenge,b} \wedge \mathsf{S}_{a,b} && (\text{“extension with } \mathbf{L}\text{”}) \\
&\Leftrightarrow (\mathsf{R}\,;\mathbf{L})_{a,b} \wedge \mathsf{S}_{a,b} && (\mathsf{T7}) \\
&\Leftrightarrow (\mathsf{R}\,;\mathbf{L} \cap \mathsf{S})_{a,b} && (\mathsf{T4}).
\end{aligned}
$$

We will refer to this “trick” as T14.

2.2 Diagram Layers

A diagram or 3D scene contains different types of information. In order to keep them separated, [BF03] adopted a layered scheme that was introduced in [RS97]. This scheme, originally introduced in the context of graph grammars, turns out to be applicable to the relational specification of visual languages as well. The basic idea is to decompose the diagram into the different ways it can be viewed.

Fig. 2.1 on the next page shows the original example out of [RS97]. In the upper left corner we see a small Entity-Relationship diagram. The other three subfigures show the different layers of information the diagram in the upper left corner contains. In the subfigure labeled with “(b)”, we see the logical structure of the diagram. Obviously two entities are connected by a relation and one of the entities has two attributes, which matches

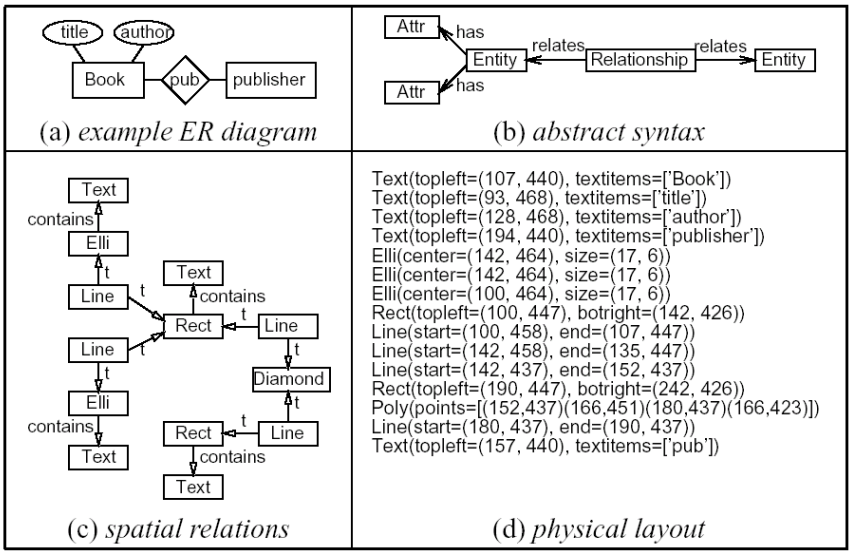


Figure 2.1: An example ER diagram [RS97] and its representations by different layers.

the original ER diagram. This view is called the *Abstract Syntax Graph* (ASG). The subfigure labeled with "(d)" shows the other "extreme", the *Physical Layout* (PL). It does not contain any logical, but purely graphical information. If the pseudo-code is transferred into some existing graphical framework, the information is sufficient to draw the diagram in "(a)". The remaining subfigure "(c)" is an intermediate view on the diagram, called the *Spatial Layout Graph* (SRG). We see that it states, for example, that two ellipses are connected to rectangles by lines, and that a diamond connects the two rectangles, also by lines. We are able to match this description to the ASG ("rectangle" corresponds to "entity", "diamond" corresponds to "relation", etc.) as well as to the Physical Layout ("rectangle" corresponds with Rect(topleft=(190,447),botright=(242,426)), etc.).

The three different views of a diagram contain a certain type of information and serve a certain purpose:

Abstract Syntax Graph: contains the logical information of a diagram without any graphical information. The ASG enables us to *interpret* the diagram.

Physical Layout: the actual geometrical information expressed in terms of coordinates, and graphical primitives as how a graphical framework would need them. The PL enables us to draw a diagram on the screen. It is the only type of information the user interacts with in an editor.

Spatial Relation Graph: a mapping between ASG and PL. It has qualitative geometrical relations instead of the quantitative description. It

does not contain any logical information but has a clear correspondence to the ASG entities.

The idea in [RS97] is to define the visual language on the level of the ASG and the SRG. In its adoption by [BF03] the graphs are considered relations. One part of our definition of J^3D_L will be to define the relations based on the ASG/SRG scheme: the logical relations of the ASG are representing concepts of the object oriented world, in this case Java. The spatial concepts of the SRG are independent of this but have to be designed to reflect 3D scenes. The connection between the layers (the correspondence between "entity" and "rectangle" in our example) will be be realized by relations.

2.3 Declarative Specification

In the last section we have shown what kinds of relations we will introduce in order to describe 3D diagrams (from now on called *scenes*) and that a layered structure of relations exits. The description of a scene alone is not a language. What is missing is a formalism that enables us to use the descriptions for defining a language.

The idea of [BF03] is to use declarative specification of the language through constraints, which goes back to work conducted by [HM91]. By declarative we mean that we decide whether or not a 3D scene is word of a language based on if it fulfills a set of properties. This differs from how grammars define a language; for which a word is part of the language if it can be constructed by rules out of terminal symbols.

The authors define picture specification languages by declaring how figures are assembled out of graphical primitives and recursively assembled out of subfigures. These definitions are constraints concerning the elements the figures are specified by. Specifications are expressed in first order predicate logic. With this approach they can both identify and construct figures.

One of their examples is a rule to specify a triangle by three lines. The syntax of these rules is that picture P is specified by subpictures $P_1, \ldots, P_n$ and constraints $R_1, \ldots, R_m$, which relate the subpictures in certain ways to express rules of the form

$$P \Rightarrow R_1, \wedge \cdots \wedge R_n[]P_1 \& \cdots \& P_n.$$

With this syntax the example can be written as:

$$
\begin{aligned}
&\mathit{triangle}(L_1, L_2, L_3) \Rightarrow \\
&\qquad \mathit{intersect}(L_1, L_2, P_1) \wedge \\
&\qquad \mathit{intersect}(L_2, L_3, P_2) \wedge \\
&\qquad \mathit{intersect}(L_3, L_1, P_3) \\
&\qquad [] \\
&\qquad \mathit{line}([P_1, P_2, P_3], \mathit{solid})
\end{aligned}
$$

Given three lines we can check whether or not they contain a triangle formed by the three intersecting points.

In a similar way we specify $\mathrm{J^3D_L}$: we start with predicate logic expressions (not following the above syntax) that describe substructures of interest in the SRG and ASG and the constraints that must hold between them. Given the elements of the scene, we can then decide whether or not it "shows the right picture", which in this case means whether or not it validly describes Java code in the correct spatial arrangement.

Deviating from the above approach we will transform the predicate logical expressions into *relation-algebraic* expressions to compute them by RelView, and we also limit ourselves to *binary* relations.

2.4 Visualization Techniques

Visualizing information three dimensionally was a research topic long before it was applied to Vise3D. Young, for example, discusses telephone networks, database structures, and version control systems, amongst others [You96]. Also, the idea to display UML diagrams in three dimensions is not new and was explored before Vise3D [Dwy01, WHF93, GRR99].

The new idea of Vise3D is to apply the visualization techniques to the three-dimensional display of object-oriented software structures. When modeling classes, for example, the idea is to not only have some objects representing classes "floating" in 3D but to use the three dimension to align the class representations according to certain schemes based on relations between the classes. The working hypothesis that goes with this approach is that the increased possibilities to arrange objects in 3D as compared to 2D allows for a better understanding of the created models. The research in this area is currently a work in progress.

We want to give a short introduction of the visualization techniques that are part of $\mathrm{J^3D_L}$. The application of these techniques is described in dedicated chapters.

Cone Trees

Cone Trees take the widely used hierarchical tree model up by one dimension. They were introduced by Robertson et al. [RMC91]. All nodes of the same tree depth are arranged on the base of a Cone with their root element on top. The Cones are rendered translucently in order to not obstruct information. When visualized, it is intended to enable the viewer to rotate any individual Cone along its longitudinal axis. By doing this, it is possible to focus on a certain piece of information without losing the context.

Although Cone Trees don't have an everyday life counterpart, the hierarchical tree is intuitively understandable and the Cone Tree allows a compact display of information in 3D.

Information Walls

Information Walls [MRC91] (aka *Perspective Walls*) can be used to display large amounts of information that otherwise would lead to large non-overlookable scenes when displayed in 2D. The obvious approach of breaking up large information quantities into separated smaller ones (e.g. scrolling or paginating) bears disadvantages for the user: by switching between different smaller bits of information, the context gets lost and the user has to reorientate.

An Information Wall is a plane in 3D space that displays the information in the same way as it would be done in 2D. Due to the effects of perspective in 3D, information that is further away fades out, which makes it easier for the human perceptive system to change focus, than in the usual 2D scrolling or switching. Details and surrounding context can be viewed at the same time, with the latter appearing farther away. Moving along a Wall smoothly turns details into context rather then cutting it off, as scrolling through a window would. Since the Wall is planar, the same layout techniques can be used for the placement of information on the Wall that were developed for 2D.

Information Cubes

The visualization technique of *Information Cubes* was proposed by Rekimoto & Green [RG93]. It visualizes nested structures as boxes that are contained in other boxes. Since the walls are rendered semi-transparently, the insides of the box are visible, thereby showing the nested structure.

The metaphor of a box containing other boxes (or other entities) is natural since such an arrangement is easily found in everyday life. Navigating through a space consisting of Information Cubes is straightforward, especially as the translucent walls allow viewing the environment while being inside the Cube. With an appropriate rendering algorithm, the inside of a

Cube might have a particular organization of nested boxes, which helps with recognizing individual cubes.

Chapter 3

Approach and Goals

We have introduced important tools and techniques in the previous chapter. Before we begin to apply them in the next chapter, we want to discuss the general approach and the goals of our thesis. In Sect. 3.1 we give a basic overview of what the language will cover. In Sect. 3.2 we describe our approach of formalizing the language and in Sect. 3.3 the goals we follow are outlined.

3.1 Outline of the Language

To get an understanding about what the language covers exactly we will describe two important aspects:

1. The object oriented concepts that can be expressed with J^3D_L.
2. The visualization of these concepts.

In terms of UML, J^3D_L covers parts of the class diagram plus the package membership diagram. The language includes the ability to model:

- classes, interfaces and packages,
- the inheritance between classes, respectively interfaces,
- the implements relationship between classes and interfaces,
- the package membership of classes and interfaces.

Other relations between classes are not covered and neither are attributes and methods. While this is a subset short of practical relevance, it will keep us busy for a long enough time.

Object-oriented programming languages differ in their concepts. Java, for example, allows only single inheritance of classes, whereas C++ allows multiple classes to be inherited. Java has a "package" concept, whereas

C++ does not. We will stick to the interpretation of these concepts as done by Java, as this is the language of choice in VISE3D.

The visualization of the Java concepts is also oriented closely at the VISE3D canon of elements. The visualization techniques that are part of J^3D_L in this thesis are (cf. Sect. 2.4):

- Cone Trees for class inheritance,
- Information Walls for interface inheritance,
- Pipes to show implements relationships between classes and interfaces.

The elements and visualization techniques are introduced in more detail in their respective chapters.

The constraints are developed to described the interaction between the graphical entities according to the requirements of the visualization techniques and the requirements to represent valid Java structures. It is not the scope of the thesis to describe how to parse the scene out of the Physical Layout, nor will it cover the extraction of information out of the source code. This kind of information is considered given for J^3D_L. Therefore, we will distinguish between two types of relations:

Definition 3.1 (Fundamental vs. Derived Relation). *If we assume that the information described by a relation is given we call the relation* fundamental. *We expect that the relation possesses all the properties we demanded for it. All other relations are built out of fundamental relations and are referred to as* derived.

When formalizing the language we will state which relations are fundamental and why we think it is valid to assume so.

3.2 The Approach

Developing J^3D_L is done in three major step. The first step is to introduce the relational "infrastructure" then we will formalize the different visualization techniques each for itself, and finally, we will integrate the three views into one. The development of constraints and the arguing that happens around their introduction constitutes the major intellectual task of the thesis. Therefore, we will introduce the scheme which we will follow to develop constraints.

3.2.1 Three Steps

To create the "infrastructure" we follow the layered approach of Rekers & Schürr. On the ASG level, the Java elements are defined and the relations between them are introduced together with the properties they have

to possess. On the SRG level, the graphical entities according to VISE3D are defined as well as their properties. To connect the layers, relations are introduced that map the Java element to the graphical entities. Some first constraints can be stated in the early stage regarding very general restrictions on the SRG level. The relations introduced in the first step are almost exclusively fundamental as they constitute the "interface" to the outer world.

Step two is to apply the relations of the first step to the individual visualization techniques. From the definition of each technique, it can be analyzed which relations need to be additionally defined and which constraints are necessary. Depending of the complexity of the visualization techniques, it might also be necessary to described substructures first and then describe their composition to form the desired structure. The constraints developed for each visualization technique fall into three groups:

Spatial constraints: ensure the correct layout of the entities according to the visualization technique.

Structural constraints: ensure the structure of the entities resemble valid Java code.

Semantical constraints: describe the correspondence of the SRG with the Java code it is supposed to represent.

The third step is to integrate the different parts of the language into one view. In the second step the constraints have been developed cumulatively, assuming that the parts were separated. The challenge in the third step is to identify the potential conflicts of displaying aspects such as package membership together with inheritance in one 3D scene. We will perform the integration in three smaller steps: first we combine Cone Trees with Cubes (*class inheritance* with *package membership* of classes), then we combine Information Walls with Cubes (*interface inheritance* with *package membership* of interfaces), and finally we analyze the integration of Cone Trees with Walls (*implements* between classes and interfaces). When conflicts are resolved, we will identify which of the earlier constraints need to be modified and how this modification is performed.

3.2.2 Developing Constraints

The development of constraints follows a similar pattern in all phases. Figure 3.1 on the following page shows an activity diagram illustrating this pattern.

The first step is a textual description of what the constraint must guarantee. The textual description describes properties a 3D scene needs to have in order to be valid geometrically or structurally. It might be the case, that the described properties are already necessarily true in a scene. In such a case we do not need to develop a relation-algebraic term that could be used

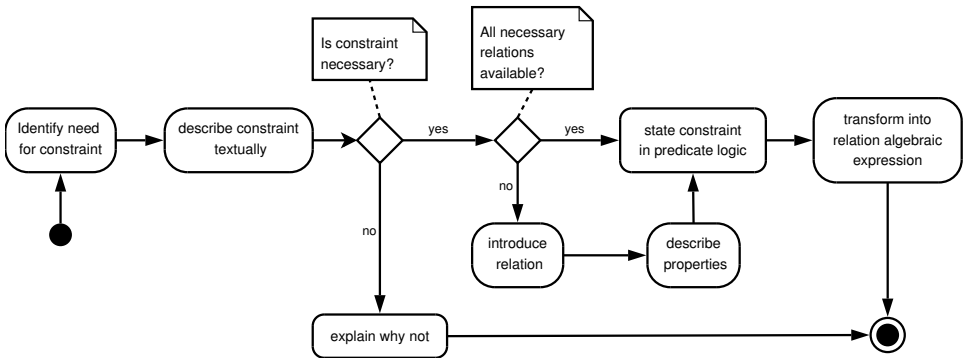


Figure 3.1: Activities involved in developing a constraint.

to check the property. Different reasons are imaginable for a required property to already be guaranteed: other constraints might imply it, the relations that are existing already necessarily bear the property, or general mathematical (e.g. geometrical) reasons imply it. In this case we will describe why we do not formalize the properties as a constraint.

If the scene does not already bear the property, we develop a relation-algebraic term that allows us to check the scene for it. Depending on the situation it might be necessary or simply convenient to develop additional relations to state the constraint. If so, they are developed. Finally the constraint can be stated, first as a predicate logical term which closely follows the textual description and then as a relation-algebraic term translated from the predicate logical statement.

3.3 Goals

In the context of Vise3D the benefit of having a visual language is that it is an implementation independent formalization of the intended visual concepts. So far, the ideas on how to visualize have been combined with implementing them in specific editors. Of course, this is valid work, especially as these implementations can serve as proof of concept and allow us to "play" with models in order to get new ideas. Any editor will have certain "short cuts" and will introduce concepts to make the usability of the editor more pleasant. The abstract definition of a visual language is free of these "flaws" and enables us to see the concepts of visualization in a more direct form.

From this perspective we can hope to answer questions such as:

- *How do the visualization techniques "work" from a conceptual level, and what are their implications?* As we decided to list any constraint, even if what the constraint states is already implied for other reasons,

we "uncover" the properties of the visualization technique independently from the chosen approach of relation algebra.

- *Is it possible to integrate the different visualization techniques, and if so, how well does the integration work?* With the conceptual description of the techniques we can answer this question from a general point of view.

- *To which degree is it feasible to delegate the properties of a scene to the language. How much should an editor support the user in creating valid 3D scenes?* This question is very important for any further development of editors. If we have a language in the form of constraints which can be stored as strings in a file and that can be fed into a relation-algebraic tool such as RelView, it is of course an ease for the developer, as no checks need to be programmed into the editor. On the other hand, it might be unrealistic to have a very general editor that is not supporting the user during creation of the scene and only depends on the separate checking of the scene. With J^3D_L, we can point out particular examples where this might not be desirable.

- *Is it realistic to check a scene by constraints, considering the amount of constraints we will have developed?* The work in [Tch04] served as a proof of concept regarding this question. The amount of constraints that were considered was low. With J^3D_L these questions can be asked again in a more realistic way.

As we picked a specific approach to define the language, we can also ask how well this approach works. In particular:

- *How well can the visualization techniques be expressed declaratively?* Through Vise3D and the research performed for each visualization techniques, we know that it is possible to create scenes by general purpose high level languages. That alone does not tell us how well the declarative description works for them. It will be interesting to see if the description is cumbersome or if the constraints get highly complicated.

- *Is the description of geometrical concepts by qualitative relations on the SRG level powerful enough to capture the requirements of the visualization techniques, where does it fail?* This is especially interesting in combination with the preceding question. In [HM91] the declarative constraints were using real number arithmetic, which makes it considerably easier to describe geometrical concepts than it works on the level of qualitative relations. On the other hand, the nature of visualization techniques seems to be that they are rather simplistic so that the description by general spatial relations might "just be right".

- *Is it problematic to be restricted to* binary *relations of relation algebra?* The binary relations are unarguably a restriction in expressiveness. The question is if this really matters when describing the scene and where it makes a difference.
- *Is the transformation from predicate logical expressions into relation-algebraic ones straightforward?* While it is possible to transfer any first order predicate logical term into relation algebra, it might turn out the the expressions we generate are only transformable for the price of using exotic relation-algebraic concepts. In such a case the question might be asked if it is a price worth paying.

As $\mathrm{J^3D_L}$ is more than a "toy language" we hope that the answers to these questions are generalizable to a certain degree.

Part II

Formalizing the Structures of the Language

Chapter 4

Language Basics

This chapter lays the ground work for all relation-algebraic formalization in the thesis. Its main purpose is to apply the three-layer separation of ASG/SRG/PL (cf. Sect. 2.2) to J^3D_L. The benefit of the three layers is a clear distinction of the different concepts. In this way we can discuss each layer separately and connect the layers afterwards. For each layer we will first introduce the necessary elements and then define the relations between the entities in the same level. These relations will be used later to phrase the constraints. The relations will be then analyzed for properties we expect them to possess.

In Sect. 4.1 we discuss the ASG, which in our case resembles the structural aspects of the Java language. For the PL, discussed in in Sect. 4.2, we will see that its formalization by means of relation algebra is possible but not desireable. Hence, the SRG, discussed in Sect. 4.3, will not make any reference to the PL and will instead introduce the spatial relations as fundamental. Having discussed each layer individually we will turn to describe how ASG and SRG connect in Sect. 4.4, which will be crucial to relate the 3D scenes to the Java code it is supposed to model. Finally we will explicitly state some assumptions we make regarding entities of the SRG in Sect. 4.5.

4.1 The Abstract Syntax Graph

The first step in adopting the Rekers & Schürr approach is to formalize the ASG.

In the case of J^3D_L, the ASG contains the information about the underlying Java elements. We will give a very short walk-through of Java elements and then describe the relations between them. The definite source concerning the structure of the Java programming language is the *Java Language Specification (JLS)* [GJS97]. As Java is an object-oriented language, readers with an OO background will find most terms and concepts familiar.

4.1.1 Entities of the ASG

The most important *non-dynamic* elements of Java include:

classes are the main organization units of the source code. They provide a scheme for creating objects of the same type and can inherit from at most one other class.

interfaces define data types as classes do, but they have no implementation. They can inherit from (multiple) other interfaces.

packages organize classes, interfaces and subpackages in analogy to file system folders.

We will treat the elements of the language as disjoint sets. When we want to make a statement, for example, about two classes, we will refer to them as two elements of the set *CLASS*. The other sets are named accordingly: *INTERFACE* and *PACKAGE*. To refer to all Java elements the set

$$JAVA := PACKAGE \cup CLASS \cup INTERFACE$$

is established as the disjoint join of the above sets. These elements can be related to each other in various ways:

Relation inheritsClass : *CLASS* ↔ *CLASS*. *This relation contains all pairs of classes where the first class inherits from the second one. Note, that in Java, unlike C++, only single inheritance is possible for classes. We define:*

$\mathsf{inheritsClass}_{c,d}$ holds if c's class definition reads: c `extends` d.

Relation inheritsInt : *INTERFACE* ↔ *INTERFACE*. *This relation contains all pairs of interfaces where the first interface inherits from the second:*

$\mathsf{inheritsInt}_{i,j}$ holds if i's interface definition reads: i `extends` j.

Relation implements : *CLASS* ↔ *INTERFACE*. *This relation contains all pairs of classes and interface where the class implements the interface. A class can implement multiple interfaces. We define:*

$\mathsf{implements}_{c,i}$ holds if c's class definition reads: c `implements` i.

Relation packageMemberOf : *JAVA* ↔ *PACKAGE*. *This relation contains all pairs of JAVA elements and packages, where the JAVA element is contained in the package. Packages can contain classes, interfaces, and subpackages. Packages and subpackages are implicitly defined by the package statements that classes and interfaces include: if, for example, a class contains the string* `package p.q`*, it is contained in package* `q`*, and* `q` *is subpackage of* `p`*. We define:*

$\mathsf{packageMemberOf}_{p,q}$ holds if p is package member of q.

Of course more than just *inheritance* exists between classes, e.g. *association* and *uses* relationships. In this thesis we will only focus on inheritsClass; other relationships between classes expose a similar structure and can be modeled alike if desired. Also, we decided to not consider nested classes; that is, classes defined in other classes. The reason for the decision is that on the one hand, they occur relatively seldom, and on the other hand, they would have required a lot of special treatment in terms of additional constraints. By studying the containment of classes and packages, respectively the visual equivalents of boxes inside cubes, (cf. Chapter 7) we will have provided a starting point for optional integration of nested classes.

In order to reference to the different types of Java elements when defining constraints, we model them as subsets of the set *JAVA*, that is for each of *CLASS*, *INTERFACE* and *PACKAGE* a vector is introduced which models the appropriate subset:

Relation class : *JAVA* ↔ 1. *This vector models the subset of JAVA elements that are classes:*

class_c holds if c is a class.

Relation interface : *JAVA* ↔ 1. *This vector models the subset of JAVA elements that are interfaces:*

$\mathsf{interface}_i$ holds if i is an interface.

Relation package : *JAVA* ↔ 1. *This vector models the subset of JAVA elements that are packages:*

$\mathsf{package}_p$ holds if p is a package.

4.1.2 Requirements for the ASG Relations

The relations come with certain requirements that stem from the Java concepts they aim to represent.

Constraint 4.1. *A class can inherit from another class not more than once:*

$$\mathsf{inheritsClass} \text{ is univalent.}$$

Constraint 4.2. *The inheritance structure for all classes must be cycle-free, that is, no class can directly or indirectly inherit from itself. Relation-algebraically this is expressed by saying that the transitive closure of* inheritsClass *does not include any pair of identical classes:*

$$\mathsf{inheritsClass}^{+} \subseteq \bar{\mathsf{I}}.$$

For inheritsInt multiple inheritance is possible, therefore Cons. 4.1 is not applicable. Hence, we only demand:

Constraint 4.3. *The inheritance structure for all interfaces must be cycle-free, that is, no interface can directly or indirectly inherit from itself:*

$$\mathsf{inheritsInt}^{+} \subseteq \bar{\mathsf{I}}.$$

For the implements relation, we have none of the above restrictions and cycles are not possible, as the heterogeneous relation only works in one direction, from classes to interfaces, and any number of classes can implement any number of interfaces.

For the packageMemberOf relation, we get:

Constraint 4.4. *No package can be a member of itself or belong to two different packages at the same time:*

$$\mathsf{packageMemberOf} \text{ is irreflexive and univalent.}$$

Constraint 4.5. *No cycles may exist in the package membership. A package can not indirectly be member of itself:*

$$\mathsf{packageMemberOf}^{+} \subseteq \bar{\mathsf{I}}.$$

4.2 Physical Layout

On of our main goals is to make $\mathrm{J^3D_L}$ independent from the Physical Layout, that is from the geometrical details of a scene. As a result, the constraints in the later sections make no use of PL-relations. Nevertheless we would be able to model the Physical Layout with relation algebra and also to introduce relations that connect the Physical Layout with the Spatial Relation Graph. We will show how such an formalization could be performed, but also argue why it is not preferable to use this model in an implementation.

4.2.1 A Relation-Algebraic Model of the J3DL Universe

First of all we define an universe for $\mathrm{J3D_L}$:

Definition 4.1 (J3DL Universe). *The universe in* $\mathrm{J3D_L}$ *is a finite set of points that constitutes an three-dimensional Euclidean space. The set of all points is named POINT.*

Fig. 4.1 shows the labeling of the three axes. Each of the points in the universe is described by three coordinates, whereas each coordinate's value is a natural number. With this arrangement we do not have a continuous but a discrete space and the points are "pixels", an appropriate granularity for a computer based 3D application.

The advantage of the discrete coordinates is that we can easily model natural numbers (including zero) in relation algebra: a structure $(N, \mathsf{zero}, \mathsf{succ})$ models the natural number if the following holds:

- $\mathsf{zero} : N \leftrightarrow 1$ is a point,
- $\mathsf{succ} : N \leftrightarrow N$ is univalent and injective,
- $\mathsf{succ}^{\mathsf{T}^*} ; \mathsf{zero} = \mathbf{L}$, which means that all elements of N are reached from the element represented by zero, and
- $\mathsf{succ} ; \mathsf{zero} = \mathbf{O}$, which means that element represented by zero is successor to no other element (which makes it the first element).

If we have $(\mathsf{succ})^{+}_{a,b}$ we know that a is an (indirect) successor of b which can be interpreted as "greater than" in terms of the natural numbers. For each coordinate we can introduce a relation that relates each point with a natural number, its coordinate value:

$$\mathsf{xCoord} : \mathit{POINT} \leftrightarrow N,$$
$$\mathsf{yCoord} : \mathit{POINT} \leftrightarrow N,$$
$$\mathsf{zCoord} : \mathit{POINT} \leftrightarrow N.$$

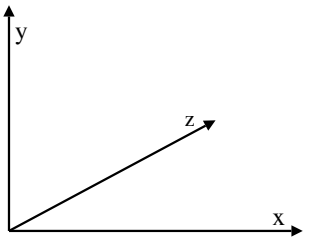


Figure 4.1: The three axes of the $\mathrm{J3D_L}$ universe and their labeling.

These relations are mappings, as each point has exactly one value associated for each coordinate. Together with succ it is easy to describe concepts such as *left of* or *above*:

Relation leftOf : ***POINT*** $\leftrightarrow$ ***POINT*.** *This relation contains all pairs of points where the first point is left of the second:*

$$\begin{aligned} \mathsf{leftOf}_{p,q} : &\Leftrightarrow \mathsf{succ}^{+}_{\mathsf{xCoord}(q),\mathsf{xCoord}(p)} \\ &\Leftrightarrow \exists x, y\ \mathsf{xCoord}_{q,x} \wedge \mathsf{xCoord}_{p,y} \wedge (\mathsf{succ})^{+}_{x,y} && \text{(T13)} \\ &\Leftrightarrow (\mathsf{xCoord}\,;(\mathsf{succ}^{+})^{\mathsf{T}}\,;\mathsf{xCoord}^{\mathsf{T}})_{p,q} && \text{(T2,T7).} \end{aligned}$$

The connection to the SRG would happen by defining the entities on the SRG levels as subsets of *POINT*. That way relations as overlap or contains are simply relation-algebraic versions of set theoretic operations as shown in [BF03].

To connect the Physical Layout with the Spatial Relation Graph we would introduce a relation that connects entities with the points they cover in space:

Relation point : ***POINT*** $\leftrightarrow$ ***ENTITY*.** *This relation relates points to the entities that allocate them as follows:*

$\mathsf{point}_{p,e}$ holds if point p is part of entity e.

With this relation we could relate the PL and the SRG. For example the *left of* property of points could be transferred to entities.

4.2.2 Reasons Against the Usage of this Approach

If the formalization is conceptually so easy to perform, why did we decide not to incorporate it into $\mathrm{J^3D_L}$? There are three important reasons:

- this formalization is not necessary,
- it is not high-performance,
- and it is complicated to provide all necessary functionality.

Our focus is to describe the language on the level of the Spatial Relation Graph. We want to achieve an abstraction from the quantitative details to a qualitative description. To state one invariant for the Cone Trees, for example, it is sufficient to express that one box is *above* others. Knowing how above is constructed by referring to the points of the space through a successor relation is no help in order to understand the geometrical challenges of Cone Trees or any other visualization technique. In addition, any 3D

framework that is used to implement an editor for the language will come with some model of the 3D space and will have some notion of entities and if they, for example, overlap. It would be redundant to model this relation-algebraically when we can safely assume that these functions will be available elsewhere.

Most likely the implementation of geometric operations by relation-algebraic formulas would be less efficient. Special purpose algorithms exist which easily outperform the relation-algebraic ones, especially since they can execute arbitrary arithmetic operations to solve geometrical equations.

Finally it takes a lot of work to compute, for example distances, which are crucial for some of the constraints. Distances between points in a natural numbered coordinate system are not necessarily natural numbers themselves and it would take a lot of auxiliary relations to "tame" such a concept, whereas virtually any 3D framework offers the required computations already.

We have seen how we could handle the geometrical space with relation algebra but the arguments that stand against it let us decide to not depend on this approach. The consequence is that we will not describe how to derive the spatial relations on the SRG level out of PL relations. The spatial relations on the SRG level will be introduced as *fundamental*. That way we are independent from any implementation decision, whether somebody decides to use the relation model to derive the SRG relations or some other approach.

In the next section we will introduce relations on the SRG level. While we do not use the relation-algebraic model for the coordinates, the general assumptions about the universe will still be valid.

4.3 Formalizing the SRG

On the level of the Spatial Relation Graph we describe the 3D scene qualitatively. We will introduce a restricted number of relations that cover intuitive concepts of spatial reasoning. First we introduce the entities themselves and then we introduce the relations between them. For these relations we will also introduce constraints which formalize the intuitive notion of the underlying concepts. This step became necessary as we consider the spatial relations as fundamental and did not derive them out of the Physical Layout. Finally we will introduce some basic constraints regarding entities.

4.3.1 The Entities on the SRG Level

Our first step will be to introduce the geometrical objects. As mentioned in Chapter 1, J^3D_L takes the concrete syntax out of the body of work that developed around [Eng00]. The atomic elements of J^3D_L are:

boxes represent instances of classes.

spheres represent instances of interfaces.

cubes represent packages; they contain classes, interfaces, and subpackages that are members of the package they represent.

pipes represent different relationships that are possible between classes and interfaces. In the case of $\mathrm{J^3D_L}$ these are $\mathsf{inherits}_{\mathsf{Class}}$, $\mathsf{inherits}_{\mathsf{Interface}}$ and $\mathsf{implements}$.

These (atomic) entities will be the building blocks of the (compound) constructs defined by the visualization techniques. Note that the Cube later on appears also as a visualization technique (Chapter 7) although it is here introduced as an atomic entity. The difference is that the cube as such is an atomic entity, which means we do not consider it to be assembled out of other parts. When we use the cube to display information in it, we make use of the visualization technique "Information Cube".

As before with the Java language elements we will collect all entities of the same type in appropriately named, disjoint sets:

$$ENTITY = BOX \cup SPHERE \cup CUBE \cup PIPE \cup CONE$$

Additionally we will represent each subset as a vector of *ENTITY* just as we did it with the language elements.

Relation box : $\boldsymbol{ENTITY \leftrightarrow 1}$. *This vector models the subset of ENTITY elements that are boxes:*

box_b holds if b is a box.

Relation sphere : $\boldsymbol{ENTITY \leftrightarrow 1}$. *This vector models the subset of ENTITY elements that are spheres:*

sphere_s holds if s is a sphere.

Relation cube : $\boldsymbol{ENTITY \leftrightarrow 1}$. *This vector models the subset of ENTITY elements that are cubes:*

cube_c holds if c is a cube.

Relation pipe : $\boldsymbol{ENTITY \leftrightarrow 1}$. *This vector models the subset of ENTITY elements that are pipes:*

pipe_p holds if p is a pipe.

4.3.2 Relations Between Entities

Next, we have to describe the relation between entities. We have already discussed that we would not consider the Physical Layout and directly start with fundamental relations on the SRG-level. We will not explicitly assume anymore that entities are subsets of *POINT*, although this should stay in the back of our minds when defining the necessary relations. To pick up our earlier example, if we introduce a relation

$$\mathsf{leftOf} : \mathit{ENTITY} \leftrightarrow \mathit{ENTITY}$$

we can imagine that it could have been derived from a relation

$$\mathsf{leftOf} : \mathit{POINT} \leftrightarrow \mathit{POINT}$$

but since we do not have a relation *POINT* we can not show how the first has been derived out of the latter. Instead, we consider the first relation as *fundamental*; that is, we assume that the implementing program provides it for us. We separate two groups of relations: the first type is directly concerning entities and the second type compares entities with reference to a coordinate system.

Entity Related

Without using coordinates we can ask whether or not two entities overlap or not. A special case of overlapping is containment. We get:

Relation overlaps : $\boldsymbol{\mathit{ENTITY} \leftrightarrow \mathit{ENTITY}}$**.** *This relation contains all pairs of entities which overlap:*

$\mathsf{overlaps}_{e,f}$ holds if parts of e and f cover the same space.

As we have not derived the relation out of the Physical Layout but consider it fundamental, we have to explicitly state which properties we assume it possesses.

Constraint 4.6. *If one entity overlaps with the other the reverse must be true, and no entity overlaps with itself:*

$\mathsf{overlaps}$ *must be symmetric and irreflexive.*

The second part of the constraint might sound odd, but we find it more helpful to exclude the trivial fact that two identical entities have some, namely all, space in common. When using this term in normal expression, this case is ignored, and also in our constraints it is more convenient to not have to exclude the case that two entities are identical. If it would be necessary to include identical entities the relation could be replaced by its reflexive closure $\mathsf{overlap} \cup \mathsf{I}$.

Relation contains : ***ENTITY*** $\leftrightarrow$ ***ENTITY*****.** *This relation contains all pairs of entities where the first contains the second:*

$\mathsf{contains}_{e,f}$ holds if f completely overlaps with e.

For contains we assume:

Constraint 4.7. *If entity e contains entity f the reverse cannot be true. If e contains f and f contains g, entity e must contain also g, but no entity can (directly or indirectly) contain itself. These properties are expressed by stating:*

contains *must be a strict partial order.*

The same train of thought applies to explain the irreflexiveness. As the relation is transitive we do not need to refer to the transitive-closure to ensure the cycle-freeness. Saying that an transitive relation is irreflexive, respectively asymmetric as in this case, is sufficient to exclude any cycles. As we demanded that the contained entity completely overlaps with the containing entity, we know that contains $\subseteq$ overlaps holds.

We need to introduce one special relation that tells us which pipe connects with which entity. In the context of the above relations this could probably best be imagined as a *touches* relation, where two entities are in the relation if they are so close to each other that moving one of them by only one point would result in an overlap. Since the only use of this relation occurs in pipes we do not introduce a general relation of this kind. Pipes do have a direction (they resemble arrows) which is important semantically. We define two relations one for the beginning of a pipe and one for the end:

Relation pipeStarts : ***PIPE*** $\leftrightarrow$ ***ENTITY*****.** *This relation contains all pairs of pipes and entities where the pipes* start *in the entities:*

$\mathsf{pipeStarts}_{p,e}$ holds if pipe p starts in entity e.

Relation pipeEnds : ***PIPE*** $\leftrightarrow$ ***ENTITY*****.** *This relation holds all pairs of pipes and entities where the pipes* end *in the entities:*

$\mathsf{pipeEnds}_{p,e}$ holds if pipe p ends in entity e.

As pipes are the 3D version of "arrows", we expect:

Constraint 4.8. *For each pipe exactly one entity exists that the pipe ends, respectively starts, in:*

pipeStarts *and* pipeEnds *must be univalent and total.*

With this requirement we ensure that the pipes work as arrows between entities (as in e.g. UML), where multiple starts or ends are not accepted. We also ensure that each pipe has starts, respectively ends, in an entity and thereby we disallow "dangling" pipes, as both relations have to be total.

Coordinate-system Related

With an coordinate system at hand we are able to compare the entities relative to the axes of three axes. With a coordinate system as shown in Fig. 4.1 on page 29 we can associate the x-axis with *left-right*, the y-axis with *above-below* and the z-axis with *behind-in front of*. We will only show the formalization for *left-right* as the other relations have absolutely identical requirements.

Relation leftOf : ***ENTITY*** $\leftrightarrow$ ***ENTITY*****.** *This relation contains pairs of entities where the first entity is left of the second one:*

$\mathsf{leftOf}_{e,f}$ holds if e is left of f.

Relation rightOf : ***ENTITY*** $\leftrightarrow$ ***ENTITY*****.** *This relation contains the pairs of entities where the first entity is right of the second one:*

$\mathsf{rightOf}_{e,f}$ holds if e is right of f.

Constraint 4.9. *For* leftOf *(analogously for* rightOf*) must be true, that if a is left of b the reverse cannot be true and no identical elements can be left of each other. If a is left of b and b is left of c it must be true that a is left of c. This can be expressed by stating:*

leftOf *and* rightOf *must be strict partial orders.*

Equally important is the fact that the two relations are connected with each other:

Constraint 4.10. *If e is left of f it must imply the f is right of e and vice versa:*

$$\begin{aligned} &\forall e, f\ \mathsf{leftOf}_{e,f} \leftrightarrow \mathsf{rightOf}_{f,e} \\ &\Leftrightarrow \mathsf{leftOf} = \mathsf{rightOf}^{\mathsf{T}} \qquad (\mathsf{T2}, \mathsf{T12}). \end{aligned}$$

The above could also be interpreted as: it is only necessary to provide one relation for each coordinate and the other can be computed. In complete analogy other spatial relations can be introduced (skipping the formal definition, for reference see rightOf and leftOf):

Relation above : ***ENTITY*** $\leftrightarrow$ ***ENTITY*****.** *This relation contains all pairs of entities where the first entity is above of the second one.*

Relation below : ***ENTITY*** $\leftrightarrow$ ***ENTITY*****.** *This relation contains all pairs of entities where the first entity is below the second one.*

Relation behind : $\boldsymbol{ENTITY} \leftrightarrow \boldsymbol{ENTITY}$**.** *This relation contains all pairs of entities where the first entity is behind the second one.*

Relation inFrontOf : $\boldsymbol{ENTITY} \leftrightarrow \boldsymbol{ENTITY}$**.** *This relation contains all pairs of entities where the first entity is in front of the second one.*

The constraints for these relations are exactly the same as for leftOf and rightOf and we will not explicitly define them.

The last relation we need in the context of the coordinate system enables us to compare distances. This does, of course, not mean that we aim to compare real-valued distance information. All we want to compare is whether a pair of entities has the same distance to each other than another pair.

Relation sameDistance : $\boldsymbol{ENTITY} \times \boldsymbol{ENTITY} \leftrightarrow \boldsymbol{ENTITY} \times \boldsymbol{ENTITY}$**.** *This relation contains all pairs of entity tuples, that have the same distance to each other:*

$\mathsf{sameDistance}_{<e,f>,<g,h>}$ holds if the distance between e and f equals the distance between g and h.

Obviously we expect:

Constraint 4.11. *The distance between the two pairs of entities is the same, even if we reverse the order; and it is also the same between two identical pairs:*

sameDistance *must be reflexive and symmetric.*

4.3.3 Basic Constraints

Without any further study of the visualization techniques and the additional constraints they bring, we can state a very basic constraint.

Constraint 4.12. *No two instances of the atomic entities box, sphere and pipe can ever overlap. If entities e and f are either box, sphere and pipe the can not be related by* overlap*:*

$$\forall e, f\ (\mathsf{box}_e \vee \mathsf{sphere}_e \vee \mathsf{pipe}_e) \wedge (\mathsf{box}_f \vee \mathsf{sphere}_f \vee \mathsf{pipe}_f) \rightarrow \overline{\mathsf{overlap}}_{e,f}$$

$$\Leftrightarrow \forall e, f\ (\mathsf{box} \cup \mathsf{sphere} \cup \mathsf{pipe})_e \wedge (\mathsf{box} \cup \mathsf{sphere} \cup \mathsf{pipe})_f \rightarrow \overline{\mathsf{overlap}}_{e,f} \qquad (\mathsf{T5})$$

$$\Leftrightarrow (\mathsf{box} \cup \mathsf{sphere} \cup \mathsf{pipe})\ ;(\mathsf{box} \cup \mathsf{sphere} \cup \mathsf{pipe})^{\mathsf{T}} \subseteq \overline{\mathsf{overlap}} \qquad (\mathsf{T2},\mathsf{T7}).$$

It is not desirable for the entities to "collide" as it has no semantical correspondence and it could confuse the user as information could be hidden. The Cube is excluded as its intended visualization is to contain other entities as discussed in Chapter 7.

4.4 Connecting ASG and SRG

We have now walked through the three different layers and seen what objects exist on the different layers (or not, as in the case of the Physical Layout) and how we will model them. The purpose of J^3D_L is to make statements about how and if the arrangement of certain entities on SRG level resembles valid Java structures. Therefore we need to establish a connection between the SRG and the ASG.

The trivial though important step is to identify each type of entity with exactly one type of Java element and to document this identification by relations.

Relation classOf : ***BOX*** $\leftrightarrow$ ***CLASS.*** *This relation contains the information about which box represents which class:*

$\mathsf{classOf}_{b,c}$ holds if box b represents class c.

Relation interfaceOf : ***SPHERE*** $\leftrightarrow$ ***INTERFACE.*** *This relation contains the information about which sphere represents which interface:*

$\mathsf{interfaceOf}_{s,i}$ holds if sphere s represents interface i.

Relation packageOf : ***CUBE*** $\leftrightarrow$ ***PACKAGE.*** *This relation contains the information about which cube represents which package:*

$\mathsf{packageOf}_{c,p}$ holds if cube c represents package p.

To ensure that the mapping between Java elements and entities works as intended we state:

Constraint 4.13. *Each Java element must be represented by exactly one geometrical entity of the appropriate type and vice versa. More precisely: Each class is related to exactly one box, each interface exactly to one sphere and each package to exactly one cube (and vice versa):*

classOf, sphereOf *and* packageOf *have to be bijective.*

The constraint ensures the most basic form of correspondence between the ASG and the SRG. It ensures that each object in a scene results in the appropriate bit of Java code. In the relational world, the above requirement translates into stating that the three relations are *bijective*.
Due to the fact that a bijection is a mapping we can write the relations classOf(), interfaceOf() and packageOf() as functions. Later on, when discussing the integration of different visualization techniques in Part III, we have to diverge from the strict bijective correspondence, but up to that point we assume that the above constraints are appropriate.

4.5 Assumptions

In order to be comprehensive in our discussion of entities we will discuss some assumptions we make about the entities and their orientation in the universe. The reason for these assumptions is *simplicity*. We consider it justified to introduce some basic assumptions that either are intuitively clear or that can be agreed upon without any "real-world" loss of expressiveness for $\mathrm{J^3D_L}$.

Assumption 4.1. *Pipes run straight between the center of the two entities they connect and their diameter is significantly smaller than the dimensions of the boxes or spheres they are connecting.*

This assumption is intuitive, because in virtually any modeling language the arrows are smaller than the objects they connect. The benefit for our effort is that we do not need to consider the pipe if we do consider the entities on the ends. If the two entities that are connected by a pipe are, for example, contained by a Cube, the pipe will be contained by the Cube also, as it is located in between the entities.

Assumption 4.2. *The point of reference of any atomic entity, which is the point used in determining the position and distance to other entities, is defined to be the* center *of the entity.*

Relations such as leftOf are fundamental; that is, not computed as part of $\mathrm{J^3D_L}$. Therefore we do not have care about what assumptions are made when they are computed. On the other hand, it is helpful in the course of the thesis to have decision regarding the reference point; that way we can sketch example situations consistently. The center of an entity seems to be a safe choice: the distance between entities does not change if they are rotated around their center axes, nor does the center change when scaling the entity.

Assumption 4.3. *All atomic entities of the same type have the same dimensions. Cubes are excluded from this assumption.*

This assumption can be helpful when aligning entities of one type along one of the axes (e.g. boxes along the x-axis). Together with Assumption 4.2 we know that none of the entities protrudes. Cubes are naturally excluded as their size depends on what they contain which can vary considerably.

Assumption 4.4. *Any assembled structure (the visualization techniques, as Cone Tree and Wall), will have at least one axis parallel to the coordinate system.*

To fully understand this assumption we would need to go into the details of the particular visualization techniques. Nevertheless, this assumption is

highly important for our "qualitative" description to work. It is not possible for us to model with our spatial relations that some subset of entities has to be arranged on a plane, if this plane is not parallel to one of the three axes. This limitation is bypassed when we can assume that we can always relate the arrangement of entities to the axes of the coordinate system.

4.6 Conclusion

At this point we would be able to model simple 3D scenes with a first (incomplete) interpretation as Java code. The formalization for each level was not very challenging. We believe this is due to two reasons:

1. The complexity was not very high, as the language elements are not interconnected yet.

2. Thanks to the Rekers & Schürr approach, respectively the idea to apply this approach to J^3D_L and to combine it with relation algebra as proposed by Berghammer & Fronk, the different layers could have been dealt with separately.

Next to the fact that "graph" and "relation" are closely related it is the intuitiveness with which Java concepts and spatial concepts are understandable that make their approach valuable. The analysis of the relations was simple as they were direct translations of what one would expect from the common concepts we used.

It will be the task of the next chapters to test if the simplicity of the basic language concepts will be sufficient to describe the visualization techniques and their integration.

Chapter 5

Cone Trees

In the VISE3D context, the Cone Trees are used to display inheritance relations between classes. If A is the inherited class for B and C, and B has the subclasses E and F, then two Cones are necessary: C_1 with A at the top and B and C at the base, and C_2 with B at the top and E and F at the base. The two Cones are arranged in a way that B is at the base of Cone one and the top of Cone two. Figure 5.1 shows a J^3D_L Cone Tree representing an inheritance structure of classes.

Upcoming in Sect. 5.1 is a concise definition of Cones and Cone Trees and their realization in J^3D_L. Afterwards we will see in Sect. 5.2 why it is important to differentiate between *potential* and *actual* Cones, how it is possible to represent Cones by a single box, and Sect. 5.3 introduces the constraints for Cones. In Sect. 5.4 we discuss the need for an entity that represents Cones. The formalization of Cone Trees in Sect. 5.5 follows a similar procedure. Finally we will discuss the semantical constraints in Sect. 5.6 and give a summary of our work in Sect. 5.7.

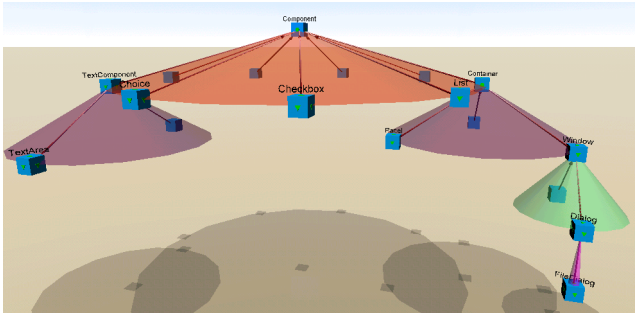


Figure 5.1: A J^3D_L Cone Tree.

5.1 Definitions and Requirements

Up to this point the reader might have a good intuition as to what a Cone Tree is. For the purpose of language definition, we will need to give a more rigid definition. Since Cone Trees do not have a *canonical* form, the particular layout largely depends on the context in which they are used. Also, there are always several ways to define geometric entities; in this case Cones, and we made special assumptions about the alignment of assembled entities, such as Cone Trees, in Sect. 4.5. We will first introduce Cones, to then explain how Cone Trees are built from Cones.

5.1.1 Cones

When talking about Cone Trees, it is important to distinguish the geometrical entity of a Cone (see Def. 5.1) from the J^3D_L version of a Cone (cf. Def. 5.2). Whenever the necessity arises to separate those two versions, we will write *g-Cone* and J^3D_L-*Cone* respectively. From the geometric point of view the g-Cone is a solid object with an infinitely large set of points, whereas the J^3D_L-Cone is a set of boxes that resemble the shape of a Cone by their layout.

Definition 5.1 (Cone, geometrical [BSMM01]). *A Cone is a pyramid with a circular cross section. The point on top is called the* vertex. *A* right Cone *is a Cone with its vertex above the center of the base, such that the angle between the base and the axis is 90 degree.*

As discussed in Sect. 4, the Cone is oriented along the y-axis of the coordinate system, standing upright (Assumption 4.4 on page 38). For our purposes, only right Cones will be of interest. Using this, we are able to define what a J^3D_L-Cone is:

Definition 5.2 (Cone, J^3D_L). *A* J^3D_L*-Cone is a set S of two or more boxes with the following properties:*

- *one box's center is on the vertex of an imaginary g-Cone, called the* root box*,*
- *the remaining boxes' centers are on the base circle of the g-Cone, called the* child boxes*,*
- *each child box on the base is connected to the root box on top by exactly one "Cone Pipe", so that the middle of the pipe's circular surface touches the middle of the two boxes,*
- *the belonging g-Cone has a radius $r > 0$.*

A g-Cone is called a proper Cone *when every class represented by a child box is inheriting from the class represented by the root box.*

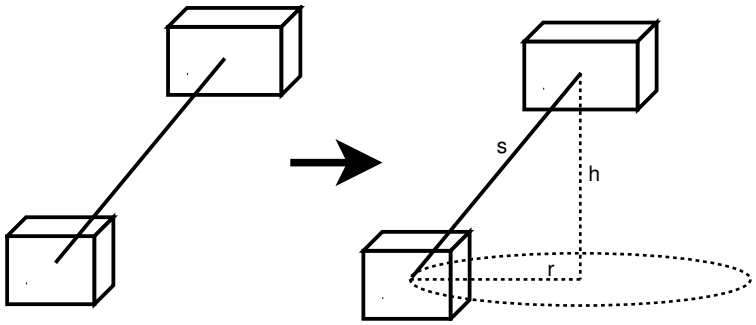


Figure 5.2: Retrieving a g-Cone out of a J^3D_L-Cone.

We will find ways later to avoid direct reference to *height* and *radius*, but for the sake of completeness we want to mention how these properties could have been deduced. The relevant insight in this context is that a J^3D_L-Cone can be related to a geometrical Cone in an unambiguous way: since the J^3D_L-Cone is aligned along the y-axis and it is a right Cone, we can deduce the radius and height from only knowing the distance s between a base box and the vertex box of the J^3D_L-Cone. Fig. 5.2 illustrates this. Therefore, we are able to talk about the height and the radius of a J^3D_L-Cone implicitly referring to the g-Cone.

5.1.2 Cone Trees

Next we define how a Cone Tree is built out of J^3D_L-Cones. Note that we will give only one definition, since we do not need to have a geometrical definition beyond the definition of g-Cones.

Definition 5.3 (Cone Tree, J^3D_L). *A set S of at least two boxes forms a Cone Tree if either S is a Cone itself or there are subsets $S_i \subset S, i \in \{1 \dots m\}, 2 \leq m \leq |S|$ so that $\bigcup_{i=1}^{m} S_i = S$ and:*

- *every S_i forms a J^3D_L-Cone,*
- *for all but one of the S_i's holds, that the box on the vertex is also a box of another J^3D_L-Cone's base circle,*
- *all J^3D_L-Cones have the same height.*

A Cone Tree is called a proper Cone Tree *if all J^3D_L-Cones in it are* proper.

The above implies, together with Def. 5.2, that a Cone Tree represents a complete Java class inheritance structure. As the reader might realize, every Java class *implicitly* inherits from the class `Object`, which will *not* be considered an inheritance in J^3D_L. The height of all Cones in a Cone Tree

must be the same, which is important to be able to oversee the different levels of hierarchy from any distance. The radii of the individual J^3D_L-Cones and the layout of the boxes on the base are not restricted to allow for an efficient layout of the "subcones," depending on their size and number of boxes.

5.2 Identifying Potential Cones

Having defined Cones and Cone Trees, we now turn to formalize the properties relation-algebraically.

Cones and Cone Trees are not part of the spatial relation graph. No atomic entity exists that could be identified with cone in the same way we identified boxes with box. Before we can formalize the constraints, it will be necessary to identify Cones and Cone Trees in a given scene. The identification is a two-step process: first, all *potential* Cones are determined, then checked to see if these potential Cones fulfill the requirements of an actual valid Cone. We will argue for the benefits of this approach after describing it in detail. Cone Trees, and therefore Cones, are set of boxes that are spatially arranged in a certain way and connected by Cone Pipes (cf. Def. 5.2). The Cone Pipe is a special type of pipe that connects each box in a Cone. In the context of J^3D_L, they are intended to represent the inherit relationship between classes.

Relation conePipe : ***PIPE*** $\leftrightarrow$ **1.** *This vector models the subset of pipes that are intended to run between boxes of a Cone:*

$\mathsf{conePipe}_p$ holds if p is a Cone Pipe.

The conePipe relation is fundamental, as it cannot be derived from any other relation in the SRG. Looking for boxes that are connected by Cone Pipes will be the main criterion for the identification of potential Cones. It is a *necessary* and *sufficient* condition for two boxes to belong to the same Cone, but that does of course not imply that the pairs of boxes connected by a Cone Pipe are the ones that are semantically intended to be connected by one, nor that their geometrical alignment in combination with other boxes is correct.

We will need to state that a box is connected to a Cone Pipe and that the pipe is *starting*, respectively *ending*, in the box. To express this we combine conePipe with the already introduced relations pipeStarts and pipeEnds.

Relation conePipeStarts : ***PIPE*** $\leftrightarrow$ ***BOX*.** *This relation contains all pairs of pipes and boxes where the pipe* starts *in the box. The Cone Pipe p*

starts at box b if p is a Cone Pipe and $\mathsf{pipeStarts}_{p,b}$ *holds:*

$$\begin{aligned}\mathsf{conePipeStarts}_{p,b} &: \Leftrightarrow \mathsf{conePipe}_p \wedge \mathsf{pipeStarts}_{p,b} \\ &\Leftrightarrow (\mathsf{conePipe} ; \mathbf{L})_{p,b} \wedge \mathsf{pipeStarts}_{p,b} && \text{(T14,T7)} \\ &\Leftrightarrow (\mathsf{conePipe} ; \mathbf{L} \cap \mathsf{pipeStarts})_{p,b} && \text{(T4).}\end{aligned}$$

Relation conePipeEnds : ***PIPE*** $\leftrightarrow$ ***BOX.*** *This relation contains all pairs of pipes and boxes where the pipe* ends *in the box. The Cone Pipe p ends at box b when p is a Cone Pipe and* $\mathsf{pipeEnds}_{p,b}$ *holds:*

$$\begin{aligned}\mathsf{conePipeEnds}_{p,b} &: \Leftrightarrow \mathsf{conePipe}_p \wedge \mathsf{pipeEnds}_{p,b} \\ &\Leftrightarrow (\mathsf{conePipe} ; \mathbf{L} \cap \mathsf{pipeEnds})_{p,b}.\end{aligned}$$

Now it is easy to state that two pipes are connected by a Cone Pipe:

Relation connectedByConePipe : ***BOX*** $\leftrightarrow$ ***BOX.*** *This relation contains all pairs of boxes that are connected by a Cone Pipe. Two boxes are connected by a Cone Pipe (in terms of this relation) if there exists a Cone Pipe p that ends in b and starts in c:*

$$\begin{aligned}\mathsf{connectedByConePipe}_{b,c} &: \Leftrightarrow \exists p\ \mathsf{conePipeEnds}_{p,b} \wedge \mathsf{conePipeStarts}_{p,c} \\ &\Leftrightarrow (\mathsf{conePipeEnds}^{\mathsf{T}} ; \mathsf{conePipeStarts})_{b,c} && \text{(T2,T7).}\end{aligned}$$

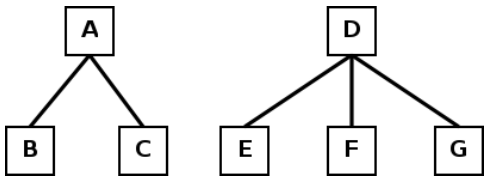
Note that the definition of connectedByConePipe has the box in which the pipe ends as the first element and the box the pipe starts out of as the second element. The direction of the pipe would suggest the reverse direction, but it seems easier to have the root-child scheme matching with the way the relation is defined above. Following this definition leaves us with all children of a root box in one row in the matrix representation of the relation.

Fig. 5.3 on the following page shows an example. We see that only the rows associated with A and D contain any entries. Later, this will be helpful when looking for an representation for Cones.

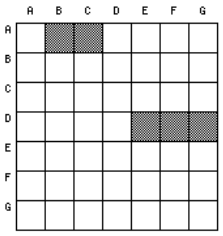
With the relations at hand, we can state our first constraints regarding the connection of boxes by Cone Pipes. The first constraint arises from the intended use of the Cone Tree visualization in $\mathrm{J^3D_L}$. The Cone Tree visualizes class hierarchies which are represented by boxes. Hence only boxes can be connected by Cone Pipes.

Constraint 5.1. *No other entities than boxes can be connected to a Cone Pipe. Whenever a pipe p is a Cone Pipe it must be true that the element related to p via* pipeStarts *is a box as well as the element related to p via* pipeEnds*:*

$$\begin{aligned}&\forall p\ \mathsf{conePipe}_p \leftrightarrow \exists b\ \mathsf{pipeStarts}_{p,b} \wedge \mathsf{box}_b \wedge \exists c\ \mathsf{pipeEnds}_{p,c} \wedge \mathsf{box}_c \\ &\Leftrightarrow \forall p\ \mathsf{conePipe}_p \leftrightarrow (\mathsf{pipeStarts} ; \mathsf{box})_p \wedge (\mathsf{pipeEnds} ; \mathsf{box})_p && \text{(T7)} \\ &\Leftrightarrow \mathsf{conePipe} = \mathsf{pipeStarts} ; \mathsf{box} \cap \mathsf{pipeEnds} ; \mathsf{box} && \text{(T4,T12).}\end{aligned}$$



(a) An example scene with two Cone Trees.



(b) The matching instance of connectedByConePipe.

Figure 5.3: Example to illustrate the decision to define connectedByConePipe in the reverse direction of the pipe's direction.

Note that the equivalence in the above constraint is only given when the Cone Pipe is the only valid type of pipe to connect two boxes with. As soon as other relationships are included in the language we have an implication instead of the equivalence.

The second constraint is due to the fact that the class hierarchies in Java are tree structures, and the Cone Pipes visualize the connection between a root and a child of the tree. It is obviously not intended for a root element of a tree to be its own child, which would happen if a Cone Pipe loop on a box would exist.

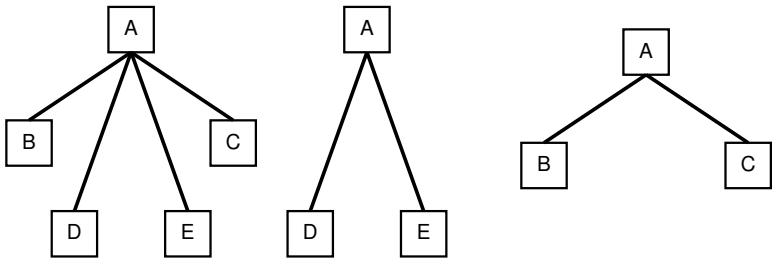
Constraint 5.2. *The boxes incident to a Cone Pipe must be* different *so no loops on one box are possible. With Constraint 5.1 we know that only boxes can be connected to Cone Pipes, all that we need to ensure here is that* connectedByConePipe *always relates different elements. This can be expressed by demanding:*

connectedByConePipe *must be irreflexive.*

As mentioned above, we will use this relation when identifying potential Cones. To keep potential and actual Cones clearly separated in our discussion, we will introduce the term *Depth One Tree (DOT)* for potential Cones.

Definition 5.4 (Depth One Tree (DOT)). *A set S of boxes is a DOT if it forms a tree of* depth one*. Exactly one box in the set exist (the* root *of this tree) that is connected to each of the remaining boxes (the* children*) by a Cone Pipe. The Cone Pipes start in the child boxes and end in the root box.*

The naming is motivated by the fact that all that is known about a potential Cone is that it has a tree structure induced by the Cone Pipes and the depth



(a) A DOT with boxes on two height-levels.

(b) The identification of two DOTs that are valid Cones.

Figure 5.4: A DOT that is not a Cone and how it would be as two Cones when applying all Requirements during identification.

of this tree is one. We know nothing about the geometrical arrangement of it at this point. It is easy to imagine examples in which the boxes are not arranged the right way, but would still be identified as potential Cones by this definition. Fig. 5.4(a) shows five boxes that are connected by Cone Pipes in the correct way, but obviously lack a spatial requirement of Cones: boxes B, C, D and E are not on the same height.

5.2.1 Benefits of the DOTs vs. Cones Approach

What are the benefits of this two-step approach? It allows us to identify all sets of boxes that *necessarily* have to be a Cone, based on the fact that they are connected by Cone Pipes. Any of the sets of boxes that follow Def. 5.4 have to fulfill all spatial constraints for Cones. By separating "DOTs" from "Cones" it is possible to phrase the difference between actual-state and target-state. Any DOT must be a Cone, and if that is not the case it can be indicated which DOT(s) do not meet one or more of the spatial constraints. This approach comes with obvious benefits that the other approaches do not carry. We want to discuss imaginable alternative approaches to work out the benefits of the DOT-Cone approach.

Alternative 1: identify only valid Cones (all structural and spatial constraints met) and then rule out the "odd" cases that might remain in the scene by special constraints.

Alternative 2: first identify potential Cones by spatial properties (instead of structural, i.e. the pipe connectivity) and then check for the required structure.

In the first case, we could not describe the deviation from the target-state as clearly without still introducing a concept of "potential" Cones. The identi-

fication is "blind" for potential-but-not-actual Cones. We would be able to identify all valid Cones and then assert that Cone Pipes are only connecting boxes of (valid) Cones. Also, we would need constraints to describe what valid Cones are *plus* constraints to ensure that no other than valid Cones in the scene exist. To us, this appears less intuitive than asserting that potential Cones are actual Cones. Another disadvantage that Alternative 1 carries is the reduced feedback it allows to give after validating a scene. It can only determine that *"Boxes A and B are connected by a Cone Pipe although not being part of any (valid) Cone"*. With the DOT-Cone approach, the validation can generate statements as *"Potential Cone XY violates the 'height' Constraint"*, which is much more specific.

With Alternative 2, it would not be ensured that any set of boxes that has the spatial Requirements for a Cone would actually be one, since it could just be an accidental arrangement of otherwise unrelated boxes. In contrast, the fact that boxes are connected by Cone Pipes according to Def. 5.4 necessarily requires them to be part of the same Cone, and if they do not meet all spatial constraints we know that this scene is not a valid $\mathrm{J^3D_L}$ scene.

5.2.2 Correctness of the DOT Identification

We now turn back to discuss why and how connectedByConePipe contains all information necessary with regard to DOTs. We will first introduce a scheme to derive DOTs by interpreting connectedByConePipe, followed by a discussion of its correctness.

For a given box r all boxes related to r via a Cone Pipes are recorded in this relation. When r is is the domain of connectedByConePipe it forms a DOT together with all boxes related to r (cf. Fig. 5.3 on page 45). Whenever r and c exists with $\mathsf{connectedByConePipe}_{r,c}$ we know that r and c belong to the same DOT and that r is the root box of that DOT and c is one of the child boxes. This scheme was the reason we defined connectedByConePipe *"reversed"*. Two questions need to be answered positively before we can depend on using connectedByConePipe as the sole source of information about DOTs:

Correctness: is everything that we interpret as a DOT using the above scheme actually a DOT?

Completeness: are all DOTs in the scene captured by connectedByConePipe so we can derive them solely by referring to this relation?

The correctness of the scheme can be answered by recalling the definition of connectedByConePipe. Any element r that appears in the domain of connectedByConePipe is incident to the end of a Cone Pipe. Each box related

to r is incident to the beginning of a Cone Pipe that ends in r. This arrangement exactly describes a DOT, which is a set of boxes connected by Cone Pipes in the above manner. Hence, the interpretation scheme yields valid DOTs.

On the other hand we need to argue for the completeness: any DOT in a scene must be found in connectedByConePipe. A DOT is consisting of at least two boxes connected by a Cone Pipe running from the child box to the root box. If more than one child box exists, more than one Cone Pipe will exist that connects to the root. In both cases, the root element will appear in the domain of connectedByConePipe, and the child box(es) in the range. Any DOT in a scene is reflected by the relation in any case. Answering both questions positively we know that connectedByConePipe allows to *correctly* discover *all* DOTs.

5.3 Cone Constraints

Now that we have identified DOTs, the next step is to describe constraints for the Cones. Each of the DOTs that have been identified must fullfill these constraints as any DOT in a valid $\mathrm{J^3D_L}$ scene must be a Cone. The constraints concern structural and spatial aspects.

5.3.1 Structural Constraints

To determine which structural constraints must be developed we will go through the Cone and Cone Tree Definitions 5.2, and 5.3 on page 42. We will use the representation scheme for DOTs from above to argue if and how the constraints must be checked. If a constraint is implicitly true, we do not need to develop a relation-algebraical expression for it. As each Cone is a DOT, we can argue about properties that DOTs have, to automatically argue about the properties a Cone must have.

Constraint 5.3. *Each Cone must have exactly one root box. This is a natural requirement for tree-like structures.*

Constraint 5.3 is implied: the uniqueness of the root box is given due to the fact that root boxes are in the domain of connectedByConePipe, and the child boxes of a DOT consist out of the boxes related to exactly one of the elements in the domain. Due to this construction a DOT, and hence a Cone, has exactly one root box.

Constraint 5.4. *Each Cone must have at least one base box. While a viewpoint is imaginable where even a single node can be considered a tree, Cones were defined to have at least two boxes.*

Constraint 5.4 is implied: every DOT also has necessarily at least one base box. For a DOT to be recognized through connectedByConePipe, at least

one Cone Pipe must exist, and therefore two different boxes (Cons. 5.2) must be in the scene that are connected by this Cone Pipe. Only then can an element be in the domain of connectedByConePipe to be considered the root of a DOT.

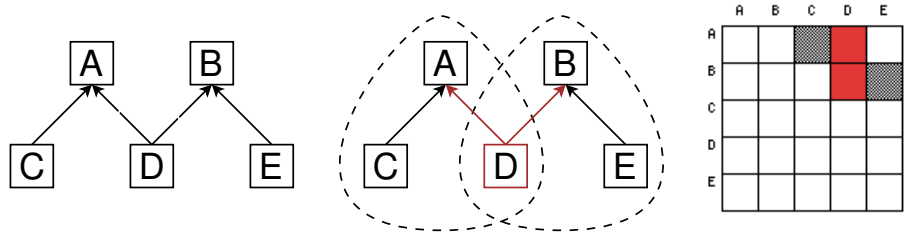
Constraint 5.5. *Each box is root box for at most one Cone. This requirement arises from the fact that only complete trees of depth one ought to be acknowledged as Cones. If a box could be root of more than one Cone, each subset of the child-boxes related to this root box would be considered a Cone. Opposingly we only want to deal with* maximal *trees of depth one.*

Constraint 5.5 is implied: to see why it is *necessarily* true that a box is the root of at most one DOT and therefore root of at most one Cone, we again argue using the domain of connectedByConePipe. A DOT is constituted by exactly one element of the domain, and all elements related to this domain element. Whenever a box is root of a DOT this can happen only one time, since it can be counted only once as part of the domain of connectedByConePipe. Only one DOT can be derived from one element as its root.

Constraint 5.6. *A box can be child box of at most one Cone. This is another requirement that is intuitively necessary for a tree-tree structure. The child boxes appear in the range of the relation. Relation-algebraically this means:*

connectedByConePipe *must be injective.*

Constraint 5.6 must be checked: the only thing that is not ensured inherently by the structure of connectedByConePipe is that a box can be part of two or more DOTs as a base box. Fig. 5.5 shows an example where box D has two outgoing Cone Pipes ending in the different root boxes. The result is that D is appearing twice in the range of connectedByConePipe and is hence part of two DOTs.



(a) Set of boxes connected by Cone Pipes.

(b) Parsing the boxes into two DOTs sharing a box.

(c) The matching instance of connectedByConePipe

Figure 5.5: Set of boxes recognized as two DOTs invalidly sharing one base box.

Note that Cons. 5.5 together with Cons. 5.6, still allows that one box is child of one and root of another DOT. This will be a requirement for the Cone Trees later on, where the Cone Trees are built out of Cones: to retrieve a complete tree from connectedByConePipe, we must pick a box that is in the domain but not in the range of connectedByConePipe. For this element we find all child-boxes and mark the parent to be on level 0 and the child boxes on level 1. We apply this process recursively: for any child box found we retrieve its children, if existing, and mark them to be on a level increased by one compared to their parent. Constraint 5.6 enforces that a child box cannot have any other parent than the one we are accessing it through, which guarantees the desired tree structure.

Representing Individual DOTs and Cones

The above shows us how connectedByConePipe suits our needs. What we have not yet discussed is how to refer to individual DOTs/Cones. Later on when dealing with Cone Trees, we want to formalize statements such as "All Cones in a Cone Tree must have the same height." Being able to enumerate and refer to the individual Cones is a necessity in this context. The most convenient - and as it turns out, sufficient - way is to identify a DOT with its *root element*. As we have seen above, the root element is unique for a DOT and with connectedByConePipe it is easy to determine the corresponding base elements. The domain of connectedByConePipe contains all DOTs (by containing all root elements) and the base boxes are the elements in the range.

The benefit is that we do not need to introduce any further relations and can stay with the compact representation of connectedByConePipe. DOTs are sets of boxes, and in a more "explicit" approach these sets would have been modeled as the Cartesian product $BOX \times BOX$ or the powerset 2^{BOX}, which would definitely be more expensive to do relational arithmetic with. Instead, we can refer to an atomic element in the already existing relation connectedByConePipe. This approach works for the purposes of this thesis. While we can imagine numerous other use cases where the root-representation is sufficient it might be necessary or more convenient to go back to a "full" representation of DOTs. This is an instance of the trade-off between the software engineering principles of *modularity* and *extensibility* versus *compactness*.

Excluding Circles

One obvious pitfall has not yet been addressed: how does the definition of DOTs or the structural Constraints that have been mentioned so far ensure that cycles are excluded? Loops on one box are already ruled out due to

Constraint 5.2, but it is still possible to connect a root box with its child. With regard to Cone Trees, even larger cycles could appear.

Constraint 5.7. *Boxes must not be connected by Cone Pipes in a cyclic manner; that is, no two boxes can exists that are directly or indirectly connected to each other twice. The constraint could be easily verified by ensuring that the transitive closure of* connectedByConePipe *does not include any pair of identical elements:*

$$(\mathsf{connectedByConePipe})^{+}_{b,c} \subseteq \bar{\mathbf{I}}.$$

In the next section we will see, how the graphical constraints will imply that these circles do not occur. The geometrical constraints are necessary for their own reasons but they have the welcome side effect of also ruling out circles of boxes connected by Cone Pipes. Exploiting this, we are able to elegantly save a constraint.

5.3.2 Geometrical Constraints

As before, we will go through the definitions for Cones (Def. 5.1 and Def. 5.2 on page 41) and develop constraints from them. This time it is not possible to save the verifying of constraints.

Constraint 5.8. *All child boxes of a Cone must have the same distance to the root box. If two "root-child" pairs of boxes <r,c> and <r,d> are each connected by a Cone Pipe c and d belong to the same Cone and r is their root box. Both pairs must have the same distance. The transformation of this constraint is lengthy, therefore we state the outcome here and show the complete transformation below:*

$$\begin{aligned}
&\forall\, r, c, d\ \mathsf{connectedByConePipe}_{r,c} \wedge \mathsf{connectedByConePipe}_{r,d} \\
&\qquad \rightarrow \mathsf{sameDistance}_{<r,c>,<r,d>} \\
&\quad \Leftrightarrow (\mathsf{connectedByConePipe}\ ;\varrho^{\mathsf{T}} \cap \pi^{\mathsf{T}})^{\mathsf{T}}\ ;(\mathsf{connectedByConePipe}\ ;\varrho^{\mathsf{T}} \cap \pi^{\mathsf{T}}) \\
&\qquad \subseteq \mathsf{sameDistance}.
\end{aligned}$$

Unfortunately, this constraint cannot be transformed in a straightforward manner, as we "get stuck". The solution is to restate the above, shifting the natural projections onto the left side. It is easy to see that:

$$\begin{aligned}
&\forall r, c, d\ \mathsf{connectedByConePipe}_{r,c} \wedge \mathsf{connectedByConePipe}_{r,d} \\
&\qquad \rightarrow \mathsf{sameDistance}_{<r,c>,<r,d>} \\
&\quad \Leftrightarrow \forall x, y\ \mathsf{connectedByConePipe}_{\pi(x),\varrho(x)} \wedge \mathsf{connectedByConePipe}_{\pi(y),\varrho(y)} \\
&\qquad \wedge\, \pi(x) = \pi(y) \rightarrow \mathsf{sameDistance}_{x,y}.
\end{aligned}$$

This way the transformation into an relation-algebraic term is possible. To get a more compact and readable transformation, we substitute connectedByConePipe with cbCP and sameDistance with sD. Then Cons. 5.8 can be transformed as follows:

$$\forall x, y\ \mathsf{cbCP}_{\pi(x),\varrho(x)} \wedge \mathsf{cbCP}_{\pi(y),\varrho(y)} \wedge \pi(x) = \pi(y) \rightarrow \mathsf{sD}_{x,y} \qquad (\text{“restating”})$$

$$\Leftrightarrow \forall x, y\ \exists r, s, c, d\ \mathsf{cbCP}_{r,c} \wedge \pi_{x,r} \wedge \varrho_{x,c} \wedge \mathsf{cbCP}_{s,d} \wedge \pi_{y,s} \wedge \varrho_{y,d} \wedge r = s \rightarrow \mathsf{sD}_{x,y} \qquad (\mathsf{T13})$$

$$\Leftrightarrow \forall x, y\ \exists r, c, d\ \mathsf{cbCP}_{r,c} \wedge \pi_{x,r} \wedge \varrho_{x,c} \wedge \mathsf{cbCP}_{r,d} \wedge \pi_{y,r} \wedge \varrho_{y,d} \rightarrow \mathsf{sD}_{x,y} \qquad (\mathsf{r} = \mathsf{s})$$

$$\Leftrightarrow \forall x, y\ \exists r\ (\mathsf{cbCP} ; \varrho^{\mathsf{T}})_{r,x} \wedge \pi_{x,r} \wedge (\mathsf{cbCP} ; \varrho^{\mathsf{T}})_{r,y} \wedge \pi_{y,r} \rightarrow \mathsf{sD}_{x,y} \qquad (\mathsf{T2,T7})$$

$$\Leftrightarrow \forall x, y\ \exists r\ \left(\mathsf{cbCP} ; \varrho^{\mathsf{T}} \cap \pi^{\mathsf{T}}\right)_{r,x} \wedge \left(\mathsf{cbCP} ; \varrho^{\mathsf{T}} \cap \pi^{\mathsf{T}}\right)_{r,y} \rightarrow \mathsf{sD}_{x,y} \qquad (\mathsf{T4})$$

$$\Leftrightarrow \forall x, y\ \left(\left(\mathsf{cbCP} ; \varrho^{\mathsf{T}} \cap \pi^{\mathsf{T}}\right)^{\mathsf{T}} ; \left(\mathsf{cbCP} ; \varrho^{\mathsf{T}} \cap \pi^{\mathsf{T}}\right)\right)_{x,y} \rightarrow \mathsf{sD}_{x,y} \qquad (\mathsf{T2,T7})$$

$$\Leftrightarrow \left(\mathsf{cbCP} ; \varrho^{\mathsf{T}} \cap \pi^{\mathsf{T}}\right)^{\mathsf{T}} ; \left(\mathsf{cbCP} ; \varrho^{\mathsf{T}} \cap \pi^{\mathsf{T}}\right) \subseteq \mathsf{sD} \qquad (\mathsf{T11}).$$

Constraint 5.9. *For any DOT it must be true that the root box is located above the children. If r and c are connected by a Cone Pipe, r is the root and must reside above c:*

$$\forall r, c\ \mathsf{connectedByConePipe}_{r,c} \rightarrow \mathsf{above}_{r,c} \Leftrightarrow \mathsf{connectedByConePipe} \subseteq \mathsf{above} \qquad (\mathsf{T11}).$$

In the next constraint, we will state that all child boxes must have the same height (y-axis). To express this we need an auxiliary relation:

Relation onSameHeight : ***ENTITY*** $\leftrightarrow$ ***ENTITY*.** *This relation contains all pairs of entities that are on the same height. Two entities b and c are on the same height if neither b is above c nor c above b. Formally we have:*

$$\begin{aligned} \mathsf{onSameHeight}_{b,c} : &\Leftrightarrow \overline{\mathsf{above}}_{b,c} \wedge \overline{\mathsf{below}}_{b,c} \\ &\Leftrightarrow \overline{\mathsf{above}_{b,c} \vee \mathsf{below}_{b,c}} \\ &\Leftrightarrow \overline{(\mathsf{above} \cup \mathsf{below})}_{b,c} \qquad (\mathsf{T5}). \end{aligned}$$

Constraint 5.10. *If two boxes b and c are connected to the same root box r they must be on the same height:*

$$\begin{aligned}
&\forall b, c\ \exists r\ \mathsf{connectedByConePipe}_{r,b} \wedge \mathsf{connectedByConePipe}_{r,c} \\
&\quad \rightarrow \mathsf{onSameHeight}_{b,c} \\
&\Leftrightarrow \forall b, c\ (\mathsf{connectedByConePipe}^{\mathsf{T}}\ ;\mathsf{connectedByConePipe})_{b,c} \\
&\quad \rightarrow \mathsf{onSameHeight}_{b,c} && \text{(T2,T7)} \\
&\Leftrightarrow \mathsf{connectedByConePipe}^{\mathsf{T}}\ ;\mathsf{connectedByConePipe} \subseteq \mathsf{onSameHeight} && \text{(T11).}
\end{aligned}$$

A Cone is required to have a radius greater than zero. As the Cones are aligned parallelly to the y axis, we must assert that no child box of a DOT has the same x- and z-axis as the root box at the same time. We introduce:

Relation onSameWidth : ***ENTITY*** $\leftrightarrow$ ***ENTITY*****.** *This relation contains all pairs of entities that are on the same width. Two entities b and c are on the same width if b is neither left of nor right of c. Formally we have:*

$$\begin{aligned}
\mathsf{onSameWidth}_{b,c} &: \Leftrightarrow \overline{\mathsf{leftOf}}_{b,c} \wedge \overline{\mathsf{rightOf}}_{b,c} \\
&\Leftrightarrow \overline{(\mathsf{leftOf} \cup \mathsf{rightOf})}_{b,c}.
\end{aligned}$$

Relation onSameDepth : ***ENTITY*** $\leftrightarrow$ ***ENTITY*****.** *This relation contains all pairs of entities that are on the same width. Two boxes b and c are on the same depth if b is neither behind nor in front of c. Formally we have:*

$$\begin{aligned}
\mathsf{onSameDepth}_{b,c} &: \Leftrightarrow \overline{\mathsf{behind}}_{b,c} \wedge \overline{\mathsf{inFrontOf}}_{b,c} \\
&\Leftrightarrow \overline{(\mathsf{behind} \cup \mathsf{inFrontOf})}_{b,c}.
\end{aligned}$$

These two relations are obviously constructed in analogy to onSameHeight. Now we can state the constraint regarding the radius:

Constraint 5.11. *The radius of a Cone cannot be zero. For each child box c must hold that it does not have the same width* and *the same depth as the root box r:*

$$\begin{aligned}
&\forall r, c\ \mathsf{connectedByConePipe}_{r,c} \rightarrow \neg(\mathsf{onSameDepth}_{r,c} \wedge \mathsf{onSameWidth}_{r,c}) \\
&\Leftrightarrow \forall r, c\ \mathsf{connectedByConePipe}_{r,c} \rightarrow \overline{\mathsf{onSameDepth}}_{r,c} \vee \overline{\mathsf{onSameWidth}}_{r,c} && \text{(T6)} \\
&\Leftrightarrow \mathsf{connectedByConePipe} \subseteq \overline{\mathsf{onSameDepth}} \cup \overline{\mathsf{onSameWidth}} && \text{(T5,T11).}
\end{aligned}$$

Correctness of the Geometrical Constraints

To see why these constraints yield exact arrangements that resemble right g-Cones following Def. 5.1 on page 41, we will go through and explain their geometrical implications one by one. Recall that the spatial relations have the *center* of entities as the point of reference. When, for example, we state that boxes have a certain distance from each other, this is measured from center to center. To illustrate these Constraints, we give diagrams that are 2D projections of 3D scenes. In these diagrams, we will consider only the x and y axis. It still should be possible to follow the arguments and to imagine appropriate 3D arrangements.

Constraint 5.8 requires the same distance of all child boxes to the root. In 3D this implies that the child boxes can be located on a sphere around the root (cf. Fig. 5.6(a)).

Constraint 5.9 turns this sphere into (almost) a hemisphere, because only those layouts are valid where the root resides above the children (cf. Fig. 5.6(b)).

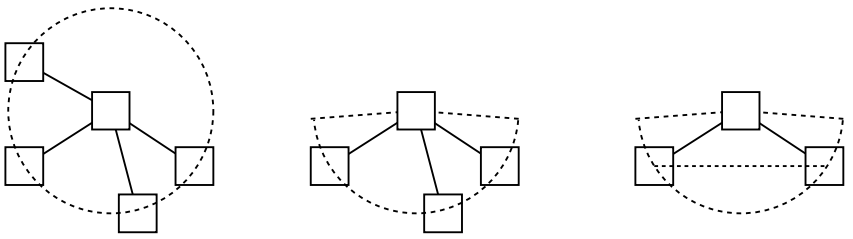
Constraint 5.10 alone enforces that the base boxes are arranged on a plane with its normal vector parallel to the y-axis. Combined with the above constraints, we have the requirement of being on the same plane as well as having the same distance to the root box. Geometrically this translates into intersecting a plane with a (hemi)sphere, resulting in a circle that has its center right below the center of the root box. This circle is the base circle of a g-Cone and the center of the root box is the vertex of that g-Cone (cf. Fig. 5.6(c)).

Constraint 5.11 ensures that the radius of the Cone's base circle is not zero, as no box can be located at the same x and y coordinates as the root.

Excluding Cycles by Geometrical Constraints

At this point, we can also argue how the spatial constraints prevent cycles. When introducing Cons. 5.7 on page 51, we said that it is not necessary to exclude cycles by checking the Cone Pipes but that the geometrical constraints would already imply the impossibility of cycles. By "cycle", we mean a situation where a Cone Pipe connects two boxes that are already indirectly connected by a path of other Cone Pipes.

The argument we will use will be able to be applied to Cones and Cone Trees alike. Fig 5.7 shows the alternatives with regard to possible cycles. Fig 5.7(a) shows a cycle with the Cone Pipes all pointing in the same direction, whereas in Fig 5.7(b) one box exists that has two Cone Pipes pointing toward

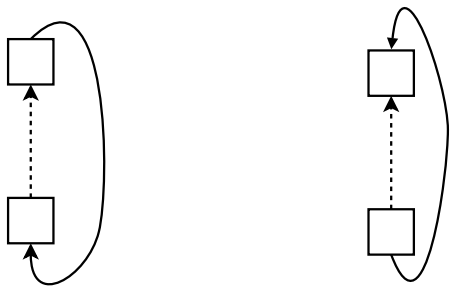


(a) Child boxes on a sphere around the root box.

(b) Base boxes restricted to lower hemisphere.

(c) Base boxes restricted to plane intersected with lower hemisphere.

Figure 5.6: Scenarios that are valid with respect to the different geometrical constraints.



(a) Cone Pipes pointing in same direction

(b) Cone Pipes pointing towards each other

Figure 5.7: Possible cycles of Cone Pipes between boxes.

it. The situation 5.7(b) can be ruled out without any further restrictions; in this scenario, a box exists that has two outgoing Cone Pipes, which is not allowed due to Constraint 5.6. Situation 5.7(a) is ruled out by the fact that Constraint 5.9 states that the vertex box of a Cone must reside above its base boxes. Due to the same directions of the Cone Pipes in this cycle, any box is a child of the box that its outgoing Cone Pipe is pointing to. As the Cone Pipes form a cycle, there has to be at least one box located above the box it is a child of. This opposes Constraint 5.9.

Definition of Cones

For the Cone *Pipe* we do not need any spatial constraints at this point, as their spatial arrangement is derived from the position of the boxes they are connecting. Therefore we now have all the bits and pieces together to give an formalized answer to the question: What is a Cone?

Definition 5.5. *A set of boxes is a Cone if it is a DOT and the Constraints 5.1 through 5.11 hold.*

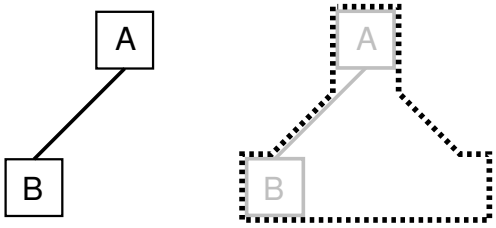
From this point on, we will assume that all DOTs are Cones and therefore not separate between them until necessary, and all further thoughts depend on the fact that all DOTs are actual Cones. For an implementation, this might mean to check the Cone constraints first to see if this assumption is true in a given scene. Otherwise, expensive computations might be made first, e.g. for Information Cubes, which could have been saved when first checking the DOT constraints.

5.4 Entities for Cones

Even when Cones and Cone Trees are not atomic entities, we will need a graphical entity for them. It is essential to detect overlapping Cones. To do so it is not enough to stipulate that the boxes and pipes of Cones do not overlap as it is already done in Chapter 4. A box could still reside within the area of another g-Cone without colliding with any of its boxes. Such an arrangement would be hard to handle for a user of an editor, when depending on the users viewpoint, a box of a Cone would be "hidden" in another Cone. Therefore we define:

Definition 5.6 (Cone Shell). *A Cone Shell is an entity that "wraps" around the g-Cone and the boxes of a Cone. The base boxes are wrapped by a disk so that the base boxes can be arranged in any way (keeping the height and radius of the Cone unchanged) without leaving the Cone Shell.*

Fig. 5.8(a) on the following page shows a Cone and Fig. 5.8(b) shows the appropriate Cone Shell. Observe how the Cone Shell covers the complete



(a) A Cone

(b) The Cone Shell around the Cone

Figure 5.8: Cone with matching Cone Shell.

g-Cone together with a disk at the bottom for all possible arrangement of the base boxes, even if they do not exist at this time. This way the base boxes can change their position relative to each other (by rearrangement) or absolutely (by rotation) without two Cones ever getting in each other's way. We assumed that boxes (and spheres) do all have the same size (Assumption 4.3), so that the height of the disk is just slightly larger than the boxes. If this assumption would not hold, the biggest base box would have to determine the size of the disk.

Cone Shells are represented by a vector coneShell as any other entity; we will skip the formal introduction (for examples refer to Chapter 4). In order to make use of the Cone Shells we need a relation that associates the Cone Shells with their Cones. This relation is, of course, fundamental as we have no way to construct the Cone Shells out of the SRG.

Relation coneShellOf : ***BOX*** $\leftrightarrow$ ***ENTITY*****.** *This relation associates root boxes (representing their Cones) with the appropriate Cone Shells:*

> $\mathsf{coneShellOf}_{b,s}$ holds if r is a root box and s is the Cone Shell of the Cone represented by r.

Recall that we decided to represent Cones by their root boxes. When implementing this fundamental relation, the base boxes of the Cone can be retrieved by connectedByConePipe. The relation is defined for all boxes, but delivers Cone Shells only for *root* boxes. Hence we must carefully ensure that we use this relation only when we know that we are dealing with root boxes. For coneShellOf to work as intended we expect:

Constraint 5.12. *Only root boxes can be associated with Cone Shells. If and only if r is a root box (in the domain of* connectedByConePipe*) a Cone*

Shells exists for r:

$$\begin{aligned}
&\forall r\ \mathsf{dom}(\mathsf{connectedByConePipe})_r \leftrightarrow \exists s\ \mathsf{coneShellOf}_{r,s} \\
&\Leftrightarrow \forall r\ \mathsf{dom}(\mathsf{connectedByConePipe})_r \leftrightarrow \mathsf{dom}(\mathsf{coneShellOf})_r \quad (\text{Def. “domain”}) \\
&\Leftrightarrow \mathsf{dom}(\mathsf{connectedByConePipe}) = \mathsf{dom}(\mathsf{coneShellOf}) \qquad (\text{T12}).
\end{aligned}$$

Constraint 5.13. *Each Cone has exactly one Cone Shell and no Cone Shell is associated by two Cones. The relation is not defined on Cones but on boxes, therefore we must restate: each box has at most one Cone Shell associated to it and no Cone Shell is associated by two different boxes. Hence we demand:*

coneShellOf *has to be univalent and injective.*

Cone Shells are hence a partial mapping, and if we know we have a root box r at hand (representing a Cone) we can write $\mathsf{coneShellOf}(r)$. With the above constraints, we know that each Cone has exactly one unique Cone Shell (through the root boxes). To state the next constraint we must describe the relationship between Cones.

Definition 5.7 (Sub-Cone). *A Cone represented by box c is* sub-Cone *of a Cone represented by box d if and only if the following holds:*

- $\mathsf{connectedByConePipe}_{d,c}$,
- $\mathsf{dom}(\mathsf{connectedByConePipe})_c$.

Box d represents a Cone due to the first requirement, and box c represents a Cone because of the second requirement. Both c and d are appearing in the domain of connectedByConePipe. The Cones are related so that c is a direct sub-Cone of d, as they are connected by a Cone Pipe. We will use this definition to state when two Cone Shells can overlap.

Constraint 5.14. *If and only if a Cone c is sub-Cone to a Cone d (c and d are the boxes box representing the Cones) their Cone Shells overlap. As* overlaps *is symmetrical we have to include the other possibility that d is sub-Cone to c. As before we substitute* connectedByConePipe *with* cbCP*:*

$$
\begin{aligned}
&\forall c, d\ \mathsf{cbCP}_{d,c} \wedge \mathsf{dom}(\mathsf{cbCP})_c \vee \mathsf{cbCP}_{c,d} \wedge \mathsf{dom}(\mathsf{cbCP})_d \\
&\quad \leftrightarrow \mathsf{overlaps}_{\mathsf{coneShellOf}(c),\mathsf{coneShellOf}(d)} \\
&\Leftrightarrow \forall c, d\ \mathsf{cbCP}_{d,c} \wedge (\mathsf{cbCP}\ ; \mathbf{L})_c \vee \mathsf{cbCP}_{c,d} \wedge (\mathsf{cbCP}\ ; \mathbf{L})_d && \text{(Def. dom)} \\
&\quad \leftrightarrow \exists s, t\ \mathsf{overlaps}_{s,t} \wedge \mathsf{coneShellOf}_{c,s} \wedge \mathsf{coneShellOf}_{d,t} && \text{(T13)} \\
&\Leftrightarrow \forall c, d\ \mathsf{cbCP}^{\mathsf{T}}_{c,d} \wedge (\mathsf{cbCP}\ ; \mathbf{L})_{c,d} \vee \mathsf{cbCP}_{c,d} \wedge (\mathsf{cbCP}\ ; \mathbf{L})^{\mathsf{T}}_{c,d} \\
&\quad \leftrightarrow (\mathsf{coneShellOf}\ ; \mathsf{overlaps}\ ; \mathsf{coneShellOf}^{\mathsf{T}})_{c,d} && \text{(T2,T14)} \\
&\Leftrightarrow \forall c, d\ \left(\mathsf{cbCP}^{\mathsf{T}} \cap \mathsf{cbCP}\ ; \mathbf{L} \cup \mathsf{cbCP} \cap (\mathsf{cbCP}\ ; \mathbf{L})^{\mathsf{T}}\right)_{c,d} \\
&\quad \leftrightarrow (\mathsf{coneShellOf}\ ; \mathsf{overlaps}\ ; \mathsf{coneShellOf}^{\mathsf{T}})_{c,d} && \text{(T4,T5)} \\
&\Leftrightarrow \mathsf{cbCP}^{\mathsf{T}} \cap \mathsf{cbCP}\ ; \mathbf{L} \cup \mathsf{cbCP} \cap (\mathsf{cbCP}\ ; \mathbf{L})^{\mathsf{T}} \\
&\quad = \mathsf{coneShellOf}\ ; \mathsf{overlaps}\ ; \mathsf{coneShellOf}^{\mathsf{T}} && \text{(T12).}
\end{aligned}
$$

When talking about Cones as entities we have to discuss one aspect that is important later on; their size (height in the case of Cones). We will discuss two alternatives: measuring the size by existing relations, or introducing a new fundamental relation to compare sizes. As it turns out, Cones are the only entities that need to be compared by their sizes in this thesis. Therefore, it is very appealing to not introduce a new relation for this one instance and the first alternative sounds very reasonable. The size (height) of a Cone could be easily be described as the distance between the vertex and the center of the base circle. On the vertex we can use the root box as a point for referral but for the center of the base circle we have no entity available. It would even be invalid to have a box at the base circle's center, as the radius of the Cone must be greater than zero (Cons. 5.11).

In order to help this situation, we could introduce a new type of entity that we use solely for the purpose of having reference points in situations like these, and therefore, it would not be intended to show such boxes in the scene the user sees. Introducing a complete new type of entity with the very special constraint of not being visible in the scene seems even more "ad hoc" as introducing a new relation that compares the sizes of entities and therefore we chose to introduce a new relation rather than introducing a new entity.

Nevertheless, the discussion illustrates one (in this thesis the *only*) situation where the avoidance of the Physical Layout and thereby the lack of points is disadvantageous. If we would have had modeled the PL we could have described the base circle's center easily without referring to any entities. After this discussion we introduce the relation to compare sizes of entities:

Relation sameSize : *ENTITY* ↔ *ENTITY*. *This relation contains all pairs of entities that have the same size. The notion of size depends on the geometrical conditions of the entity:*

$\mathsf{sameSize}_{e,f}$ holds if entity e has the same size as entity f.

Constraint 5.15. *If e has the same size as f we expect the reverse and also that e has the same size as itself:*

sameSize *must be reflexive and symmetrical.*

This relation is fundamental; as discussed above, we cannot derive it from anything in SRG. In this thesis we will solely use it to compare the height of Cone Shells. Saying that, we expect:

Assumption 5.1. *The* size *of a Cone Shell is measured by its height, especially in the context of the* sameSize *relation.*

5.5 Cone Trees

For the Cone Trees, a similar train of thought applies as for the Cones. We first need to identify potential Cone Trees by their structure; to then phrase constraints with which we can determine whether or not such a potential Cone Tree is actually a valid Cone Tree. The arguments for this two-step process are the same as they were for the Cones. If the spatial constraints were also used to identify Cone Trees, situations as depicted in Fig. 5.9 might occur, where (incorrect) Cone Trees with Cones of different heights would be recognized as separate Cone Trees, clustered by Cones of the same height. We consider this unintuitive, and therefore disregard this approach.

To distinguish between the *potential* from the *actual* Cone Tree (similar to DOT vs. Cone) we refer to potential Cone Trees as *TREES* and to the actual Cone Trees as *Cone Trees*. Each TREE is assembled of valid Cones, not just DOTs.

When we considered potential Cones having the structure of trees with depth one, potential Cone Trees would be trees with no depth restrictions. Being connected by Cone Pipes is again the necessary condition for boxes to belong to the same Cone Tree. This time, it is possible that this connection is indirect; i.e., several boxes with Cone Pipes might lay between two boxes of the same Cone Tree.

We will introduce two relations regarding TREES. Both are derived from connectedByConePipe, but they differ in how they represent the TREES. The first alternative relates Cones that are directly sharing a box.

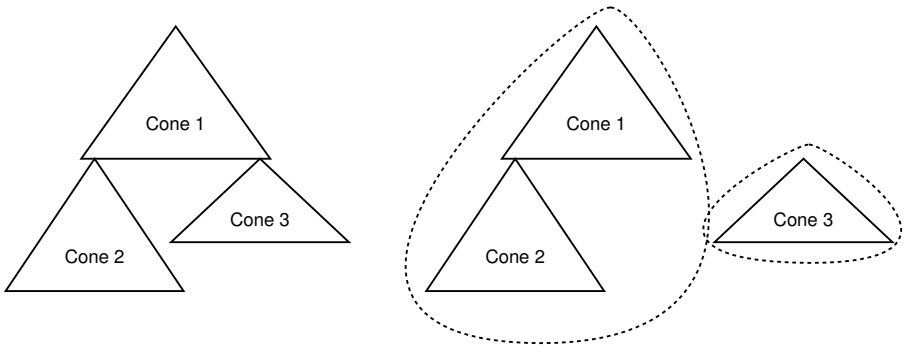


Figure 5.9: An incorrect Cone Tree and its unintuitive parsing. This approach will not be followed.

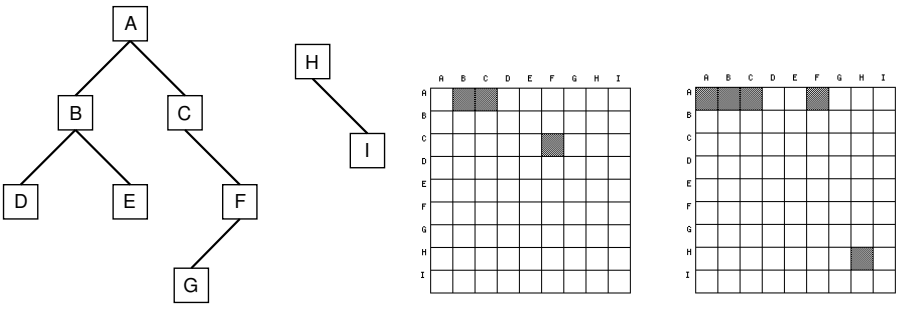
Relation coneParent : $\boldsymbol{BOX \leftrightarrow BOX}$**.** *This relation contains all pairs of Cones where the second Cone is sub-Cone of the first one. Cones are represented by their root boxes. When c is a sub-Cone of b the Cones represented by b and c belong to the same Cone Tree:*

$$
\begin{aligned}
&\mathsf{coneParent}_{b,c} \\
&:\Leftrightarrow \mathsf{connectedByConePipe}_{b,c} \wedge \mathsf{dom}(\mathsf{connectedByConePipe})_c \\
&\Leftrightarrow \mathsf{connectedByConePipe}_{b,c} \wedge (\mathsf{connectedByConePipe} ; \mathbf{L})_c \qquad (\text{Def. dom}).
\end{aligned}
$$

Now we replace the universal relation $\mathbf{L} : BOX \leftrightarrow \mathbf{1}$ with $\mathbf{L} : BOX \leftrightarrow BOX$ and we will relate box c with box b, as it is helpful later:

$$
\begin{aligned}
&\Leftrightarrow \mathsf{connectedByConePipe}_{b,c} \wedge (\mathsf{connectedByConePipe} ; \mathbf{L})_{c,b} \\
&\Leftrightarrow \mathsf{connectedByConePipe}_{b,c} \wedge (\mathsf{connectedByConePipe} ; \mathbf{L})^{\mathsf{T}}_{b,c} \qquad (\mathsf{T2}) \\
&\Leftrightarrow \left(\mathsf{connectedByConePipe} \cap (\mathsf{connectedByConePipe} ; \mathbf{L})^{\mathsf{T}}\right)_{b,c} \qquad (\mathsf{T4}).
\end{aligned}
$$

This relation does not consider TREEs consisting out of a single Cone, as was explicitly allowed by Def. 5.3. Such a TREE has no Cone that could be a parent Cone to some other, so no relation between two DOTs of this kind can be established. It gives us all Cone siblings and the parent. Whenever two Cones k and l are related via coneParent, we know that they are two root boxes that are connected by a Cone Pipe, and hence coneParent $\subseteq$ connectedByConePipe. Additionally, we know that k and l are root boxes of two Cones that *share* the box l. Fig. 5.10(a) shows an example with two Cone Trees (one of them consisting out of only one Cone) and Fig. 5.10(b) shows the belonging instance of coneParent. We see that A has two direct sub-Cones B and C, and that C itself has F as a sub-Cone. The Cone represented by H is not appearing, as it has no sub-Cone. The coneParent relation encapsulates the idea we developed when describing Constraint 5.14 on page 58, and we can restate this constraint with the new relation as:



(a) Two Cone Trees

(b) The matching instance of coneParent

(c) The matching instance of sameTree

Figure 5.10: Cone Trees and their representation by coneParent and sameTree.

Constraint 5.16 (Restating Cons. 5.14). *Constraint 5.14 stated that two Cones Shells can only overlap if the two Cones are part of the same Cone Tree. With* coneParent *the formal definition of the constraint reads:*

$$\begin{aligned}
&\mathsf{connectedByConePipe}^{\mathsf{T}} \cap \mathsf{connectedByConePipe} \,;\mathbf{L} \\
&\quad \cup \mathsf{connectedByConePipe} \cap (\mathsf{connectedByConePipe} \,;\mathbf{L})^{\mathsf{T}} \\
&\quad = \mathsf{coneShellOf} \,;\mathsf{overlaps} \,;\mathsf{coneShellOf}^{\mathsf{T}} && \text{(Def. Cons. 5.14)} \\
&\Leftrightarrow \mathsf{coneParent}^{\mathsf{T}} \cup \mathsf{coneParent} \\
&\quad = \mathsf{coneShellOf} \,;\mathsf{overlaps} \,;\mathsf{coneShellOf}^{\mathsf{T}} && \text{(Def. coneParent).}
\end{aligned}$$

The second alternative is a more direct representation of TREEs. It relates all Cones of one TREE directly to the top-level Cone of this TREE. The top Cone is represented by its root box t as any other Cone, but its unique property is that it appears in the domain but not the range of connectedByConePipe. To appear in the domain is the necessary property for all root boxes, based on the way we defined connectedByConePipe, and when it is not in the range of this relation it means that no other box exists that is a root box for t. This means that no Cone exists above the one represented by t. We will exploit this insight when defining the next relation.

Relation sameTree : $\boldsymbol{BOX \leftrightarrow BOX}$. *A Cone Tree consists out of the top Cone and all other Cones related to the top Cone. Since boxes represent the Cones, we again argue with* connectedByConePipe. *Box c represents a sub-Cone of the same Cone Tree as t if t and c are related via the transitive reflexive closure of* connectedByConePipe *and c is in the domain of* connectedByConePipe *(being a root box). Box t is the root box of a top Cone if it is in the range but not in the domain of* connectedByConePipe. *In the*

following transformation we will replace connectedByConePipe *by* cbCP *for the sake of readability:*

$$\begin{aligned}\mathsf{sameTree}_{t,c} : \Leftrightarrow\ & (\mathsf{cbCP})^*_{t,c} \wedge \mathsf{dom}(\mathsf{cbCP})_c \wedge \overline{\mathsf{ran}(\mathsf{cbCP})}_t \\ \Leftrightarrow\ & (\mathsf{cbCP})^*_{t,c} \wedge (\mathsf{cbCP} ; \mathbf{L})_c \wedge \overline{(\mathsf{cbCP}^\mathsf{T} ; \mathbf{L})}_t \qquad (\text{Def. dom, ran}).\end{aligned}$$

As done for coneParent we replace $\mathbf{L} : BOX \leftrightarrow \mathbf{1}$ with $\mathbf{L} : BOX \leftrightarrow BOX$

$$\begin{aligned}\Leftrightarrow\ & (\mathsf{cbCP})^*_{t,c} \wedge (\mathsf{cbCP} ; \mathbf{L})_{c,t} \wedge \overline{(\mathsf{cbCP}^\mathsf{T} ; \mathbf{L})}_{t,c} \\ \Leftrightarrow\ & (\mathsf{cbCP})^*_{t,c} \wedge (\mathsf{cbCP} ; \mathbf{L})^\mathsf{T}_{t,c} \wedge \overline{(\mathsf{cbCP}^\mathsf{T} ; \mathbf{L})}_{t,c} \qquad (\text{T2}) \\ \Leftrightarrow\ & \left(\mathsf{cbCP}^* \cap (\mathsf{cbCP} ; \mathbf{L})^\mathsf{T} \cap \overline{(\mathsf{cbCP}^\mathsf{T} ; \mathbf{L})}\right)_{t,c} \qquad (\text{T4}).\end{aligned}$$

sameTree represents TREEs in a similar way that connectedByConePipe represents DOTs/Cones. For each TREE we find one element in the domain, and for each related Cone one element in the range of sameTree. The one remarkable difference is the fact that the top-level Cone is part of the domain as well as the range for the same TREE. This is needed to account also for TREEs with just one Cone. The example of Fig. 5.10 has the matching instance of sameTree in Fig. 5.10(c). We see that A is a Cone Tree with three additional Cones B, C, F, but we do not see that F is a sub-Cone of C. As expected, we see the Cone Tree with only one Cone Tree, H.

Comparing the two relations, we see that coneParent shows the direct "sub-Cone" relations between Cones and sameTree shows all Cones of one Cone Tree but lacks the hierarchical information. When we textually defined the Cone Trees in Def. 5.3 on page 42, we said that the one geometrical requirement for Cone Trees is that all Cones in it have to have the same size, and as we will see for this constraint coneParent is sufficient. Again, it needs to be emphasized that only because the use of coneParent is sufficient for this thesis is it still imaginable that further extensions might require the use of relation such as sameTree.

5.5.1 Structural Constraints

We will not need to introduce any structural constraints. We represent Cones through boxes and coneParent is a subset of connectedByConePipe. Therefore the same arguments apply for Cones as they did for boxes. No Cone can be part of more than one Cone Tree in coneParent, as no box can have be part of more than one Cone in connectedByConePipe. In the same way the other structural properties translate. Of course, coneParent has to be injective, as no Cone should have two Cone parents, and we are able to prove this:

Theorem 5.1. coneParent *is injective, given that* connectedByConePipe *is injective (Cons. 5.6 on page 49).*

Proof. We need to show that $\mathsf{coneParent} ; \mathsf{coneParent}^\mathsf{T} \subseteq \mathsf{I}$ holds. To conserve space and ensure a better readability we write cbCP for connectedByConePipe:

$$
\begin{aligned}
&\mathsf{coneParent} ; \mathsf{coneParent}^\mathsf{T} \\
&= \left(\mathsf{cbCP} \cap (\mathsf{cbCP} ; \mathsf{L})^\mathsf{T}\right) ; \left(\mathsf{cbCP} \cap (\mathsf{cbCP} ; \mathsf{L})^\mathsf{T}\right)^\mathsf{T} && \text{(Def. coneParent)} \\
&= \left(\mathsf{cbCP} \cap (\mathsf{cbCP} ; \mathsf{L})^\mathsf{T}\right) ; \left(\mathsf{cbCP}^\mathsf{T} \cap (\mathsf{cbCP} ; \mathsf{L})\right) && \text{(A9,A5)} \\
&\subseteq \left(\mathsf{cbCP} \cap (\mathsf{cbCP} ; \mathsf{L})^\mathsf{T}\right) ; \mathsf{cbCP}^\mathsf{T} \\
&\quad \cap \left(\mathsf{cbCP} \cap (\mathsf{cbCP} ; \mathsf{L})^\mathsf{T}\right) ; (\mathsf{cbCP} ; \mathsf{L}) && \text{(A3)} \\
&\subseteq \mathsf{cbCP} ; \mathsf{cbCP}^\mathsf{T} \cap (\mathsf{cbCP} ; \mathsf{L})^\mathsf{T} ; \mathsf{cbCP}^\mathsf{T} \\
&\quad \cap \left(\mathsf{cbCP} \cap (\mathsf{cbCP} ; \mathsf{L})^\mathsf{T}\right) ; (\mathsf{cbCP} ; \mathsf{L}) && \text{(A3)} \\
&\subseteq \mathsf{cbCP} ; \mathsf{cbCP}^\mathsf{T} \\
&\subseteq \mathsf{I} && \text{(Assumption).}
\end{aligned}
$$

□

5.5.2 Geometrical Constraints

Only one geometrical constraint needs to be considered:

Constraint 5.17. *All Cones of the same Cone Tree must have the same height. We can compare sizes of entities with* sameSize*. Whenever two boxes are related by* coneParent *we know they are root boxes and therefore Cone Shells exist for them. We also know that the Cones they represent are part of the same Cone Tree, and therefore their Cone Shells must have the same size:*

$$
\begin{aligned}
&\forall p, c\ \mathsf{coneParent}_{p,c} \rightarrow \mathsf{sameSize}_{\mathsf{coneShellOf}(p),\mathsf{coneShellOf}(c)} \\
&\quad \Leftrightarrow \forall p, c\ \mathsf{coneParent}_{p,c} \rightarrow \exists x, y\ \mathsf{sameSize}_{x,y} \wedge \mathsf{coneShellOf}_{p,x} \\
&\qquad \wedge \mathsf{coneShellOf}_{c,y} && \text{(T13)} \\
&\quad \Leftrightarrow \forall p, c\ \mathsf{coneParent}_{p,c} \rightarrow (\mathsf{coneShellOf} ; \mathsf{sameSize} ; \mathsf{coneShellOf}^\mathsf{T})_{p,c} && \text{(T7)} \\
&\quad \Leftrightarrow \mathsf{coneParent} \subseteq \mathsf{coneShellOf} ; \mathsf{sameSize} ; \mathsf{coneShellOf}^\mathsf{T} && \text{(T11).}
\end{aligned}
$$

Now it is easy to state what a Cone Tree is:

Definition 5.8. *A set of boxes is a Cone Tree if it is a TREE and Constraint 5.17 holds.*

As one can see there is no need for a direct reference to individual TREEs; the parent-child relationship is sufficient and recursively enforces the same height. If the need would arise, it would be easier to modify Constraints 5.17 to use sameTree instead of coneParent.

5.6 Semantical Interpretation

Cone Trees were introduced to show Java class inheritance hierarchies. When we want to interpret a given scene with Cone Trees in it, we need to make sure that only those boxes are part of the same Cone Tree that actually are part of a common inheritance hierarchy of the underlying source code.

Constraint 5.18. *Exactly if class c inherits from class d must a Cone Pipe exist that runs between boxes e and f representing c and d:*

$$
\begin{aligned}
&\forall c, d\ \mathsf{inheritsClass}_{c,d} \leftrightarrow \exists e, f\ \mathsf{classOf}(e) = c \\
&\qquad \wedge\ \mathsf{classOf}(f) = d \wedge \mathsf{connectedByConePipe}_{f,e} \\
&\Leftrightarrow \forall c, d\ \mathsf{inheritsClass}_{c,d} \leftrightarrow \exists e, f\ \mathsf{classOf}^{\mathsf{T}}_{c,e} \\
&\qquad \wedge\ \mathsf{classOf}_{f,d} \wedge \mathsf{connectedByConePipe}^{\mathsf{T}}_{e,f} && \text{(T13,T2)} \\
&\Leftrightarrow \forall c, d\ \mathsf{inheritsClass}_{c,d} \\
&\qquad \leftrightarrow (\mathsf{classOf}^{\mathsf{T}}\ ;\mathsf{connectedByConePipe}^{\mathsf{T}}\ ;\mathsf{classOf})_{c,d} && \text{(T7)} \\
&\Leftrightarrow \mathsf{inheritsClass} = \mathsf{classOf}^{\mathsf{T}}\ ;\mathsf{connectedByConePipe}^{\mathsf{T}}\ ;\mathsf{classOf} && \text{(T12).}
\end{aligned}
$$

This constraint formalizes the notion of *proper* Cones as well as *proper* Cone Trees of Definitions 5.2 and 5.3. It is the only one that connects the Spatial Relation Graph with the Abstract Syntax Graph in the context of Cones and Cone Trees.

5.7 Conclusion

We have finished adapting the visualization technique of Cone Trees for their usage in $\mathrm{J^3D_L}$. After detailed description of their properties, we introduced the basic relations for their description. Since the Cone Trees are not directly available in the SRG, we described a method to identify them. As it turned out, the identification included two steps: during the first step, all potential Cones (DOTs) were identified to check then in the second step if the constraints hold that make the DOTS actual, valid Cones. As the analysis of structural constraints showed, we found a compact representation of Cones in connectedByConePipe. For all constraints, it was possible to make use of this relation. By representing Cones by their root boxes we found a compact way of arguing about Cones when referring to boxes, which laid ground for the "success" of connectedByConePipe.

The only point where the Cones could not be represented by their boxes (or pipes) was the "Cones must not overlap" requirement of Constraint 5.14. At this point, we had to introduce a completely new entity that wraps around the Cones and their boxes. Fortunately, we were able to use this new

entity also for the measurement of the Cones height so that its introduction lessened the aura of an "ad hoc" solution.

The formalization of Cone Trees followed similar thoughts, and we derived coneParent from connectedByConePipe. While this relation was not directly representing Cone Trees, it was sufficient to describe the only geometrical constraint regarding the height of Cones in the same Cone Tree. We introduced an alternative relation sameTree that represented the Cones in a Cone Tree differently and said that this relation might be helpful when formalizing additional aspects of Cone Trees. The semantical constraint was also easily formalized, again using the omnipresent connectedByConePipe.

The Cone Trees exposed much conceptual complexity that was handled gracefully by the relation-algebraic approach. We will be able to reuse many of our thoughts for the next visualization technique, the Information Walls.

Chapter 6

Information Walls

Information Walls in J^3D_L are used to display inheritance structures for interfaces. Due to the fact that interfaces allow multiple inheritance, Cone Trees are not applicable as visualization metaphors. Yet we will be able to reuse thoughts of the previous chapter.

We will first detail what Information Walls are in the context of J^3D_L in Sect. 6.1. With this understanding, we will define necessary relations in Sect. 6.2 and then discuss the need to identify potential Walls in Sect. 6.3. The constraints for the Walls will be presented in Sect. 6.4. Sect. 6.5 briefly discusses how to introduce a "Wall Shell", similar to the "Cone Shell", and the semantical interpretation of Walls can be found in Sect. 6.6. In all sections, it will be of foremost interest to compare the formalization of Walls with the formalization of Cone Trees. In Sect. 6.7 we will give a general conclusion and report on the outcome of this comparison.

6.1 Information Walls in J^3D_L

In J^3D_L the information "projected" on the Walls are the spheres representing Java interfaces and their inheritance relationship represented through pipes, whenever an *extends* relationship needs to be displayed. The Walls are obviously intended to be planar so the spheres must be arranged on the same plane. Following Assumption 4.4, the plane will be parallel to either the x-y, y-z or x-z plane.

What remains to be defined is which spheres belong to the the same Wall. Semantically, we want to show inheritance structures of interfaces. We will consider each set of interconnected spheres to be a separate Wall. In Fig. 6.1, we see several interfaces and their inheritance relations. The interfaces are grouped into three clusters. As we can see, two interfaces belong to the same cluster if the one extends the other; that is, if they are connected by an inheritance arrow. Each cluster is displayed by one Wall in J^3D_L. Structurally, the spheres and pipes can be thought of as the edges and

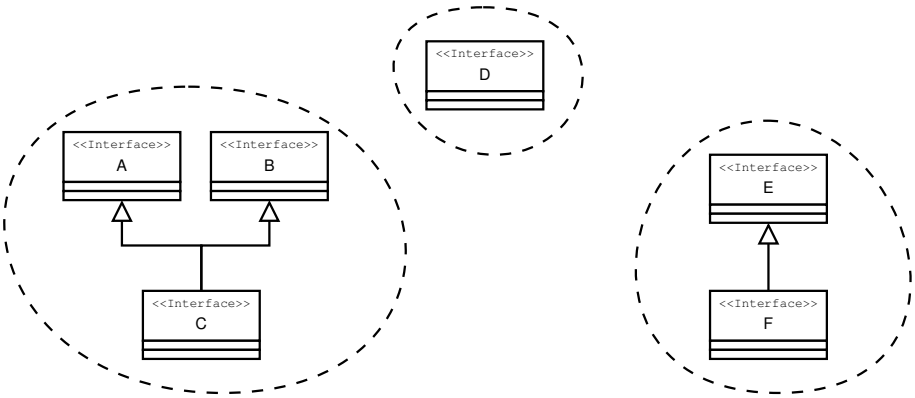


Figure 6.1: Example of interface inheritance and the clustering of interfaces into inheritance cluster.

vertices of a graph respectively. Every Information Wall then corresponds with one connection component of the graph, because the spheres that are directly or indirectly connected by pipes stand for interfaces that belong to the same Wall.

According to the Java Language Specification [GJS97], no cyclic inheritance structures are allowed. Graph-theoretically this translates into asserting that the graph is a *Directed Acyclic Graph (DAG)*. We summarize our description of Walls in $\mathrm{J^3D_L}$:

Definition 6.1 (Information Walls in $\mathbf{J^3D_L}$). *A set S of spheres is a Wall if:*

- *S has at least two elements,*
- *the graph consisting out of spheres of S as vertices and the Wall Pipes between them as edges is a DAG and consists out of a single connection component,*
- *all spheres have either the same depth, or the same height, or the same width; in other words they reside on either the same x-y, y-z or x-z plane.*

A Wall is proper *if exactly those spheres are connected by a Wall Pipe, whose interfaces are participating in the inheritance relation.*

Single spheres are not considered Walls just as single boxes are not considered Cone Trees. With this understanding of Information Walls in $\mathrm{J^3D_L}$, we now focus on the development of necessary relations.

6.2 Necessary Relations

As with Cone Trees, Walls are not part of the SRG. This time we can revert to the experiences from the formalization of Cone Trees and we will

try to apply as much as possible. From the Cone Trees, we learned that the connectivity of boxes through pipes was the crucial criterion to identify the non-atomic Cones. We also argued why it would be beneficial to first identify an intermediate, potential Cone (the DOT) to then check if it fulfills all requirements. Both will be helpful in the context of Information Walls.

Walls are assembled out of spheres and connected by a special type of pipes, called *Wall Pipes*. Before we define an appropriate "connected by" relation for the spheres, we need to distinguish the Wall Pipes from other pipes and to combine this with the information that a pipe starts or ends in a sphere. The relations are all analogous to those for Cone Trees. Hence, we do not show the complete transformations of the relation-algebraic expressions.

Relation wallPipe : ***PIPE*** $\leftrightarrow$ **1.** *This vector models the subset of pipes that are intended to run between spheres of a Wall:*

$\mathsf{wallPipe}_p$ holds if p is a Wall Pipe.

Recall that Constraint 4.8 generally prohibits "dangling" pipes. Combined with pipeStarts and pipeEnds we get:

Relation wallPipeStarts : ***PIPE*** $\leftrightarrow$ ***SPHERE.*** *This relation contains all pairs of pipes and spheres where the pipe starts in the sphere. The Wall Pipe p starts at sphere s when p is a Wall Pipe and* $\mathsf{pipeStarts}_{p,s}$ *holds:*

$$\begin{aligned}\mathsf{wallPipeStarts}_{p,s} &: \Leftrightarrow \mathsf{wallPipe}_p \wedge \mathsf{pipeStarts}_{p,s} \\ &\Leftrightarrow (\mathsf{wallPipe}\,;\mathsf{L} \cap \mathsf{pipeStarts})_{p,s}.\end{aligned}$$

Relation wallPipeEnds : ***PIPE*** $\leftrightarrow$ ***SPHERE.*** *This relation contains all pairs of pipes and spheres where the pipe ends in the sphere. The Wall Pipe p ends at sphere s when p is a Wall Pipe and* $\mathsf{pipeEnds}_{p,s}$ *holds:*

$$\begin{aligned}\mathsf{wallPipeEnds}_{p,s} &: \Leftrightarrow \mathsf{wallPipe}_p \wedge \mathsf{pipeEnds}_{p,s} \\ &\Leftrightarrow (\mathsf{wallPipe}\,;\mathsf{L} \cap \mathsf{pipeEnds})_{p,s}.\end{aligned}$$

Now we can express that two spheres are connected by a Wall Pipe.

Relation connectedByWallPipe : ***SPHERE*** $\leftrightarrow$ ***SPHERE.*** *This relation contains all pairs of entities that are connected by a Wall Pipe. Two spheres s and t are connected by a Wall Pipe (in terms of this relation) if there exists a Wall Pipe p that ends in s and starts in t:*

$$\begin{aligned}\mathsf{connectedByWallPipe}_{s,t} &: \Leftrightarrow \exists p\ \mathsf{wallPipeEnds}_{p,s} \wedge \mathsf{wallPipeStarts}_{p,t} \\ &\Leftrightarrow (\mathsf{wallPipeEnds}^{\mathsf{T}}\,;\mathsf{wallPipeStarts})_{s,t}.\end{aligned}$$

Note that we again used the *reversed* definition for connectedByWallPipe. For two spheres s, t with $\mathsf{connectedByWallPipe}_{s,t}$ the pipe starts at t and ends in s. The reason to define the relation this way is, to keep it analogous to connectedByConePipe: it was necessary to define connectedByConePipe reversely to the direction of the Cone Pipes, as we could elegantly represent Cones with this relation. The root boxes of Cones appear in the domain of connectedByConePipe and the child boxes appear in the range related to their root boxes. For Wall Pipes such an representation will not be possible nor necessary, but we want to keep the definitions of the different "connectedBy" relations as similar as possible.
In Fig. 6.2, we see a network of interconnected spheres connected by Wall Pipes. At this point we do not need to care about the geometrical interpretation. We can see one connection component $\{A, B, C, D\}$ that includes a cycle and the single sphere E. Next to the diagram we see an instance of connectedByWallPipe for this example and as expected the single sphere does not show in the relation.

6.3 Identification of Potential Walls

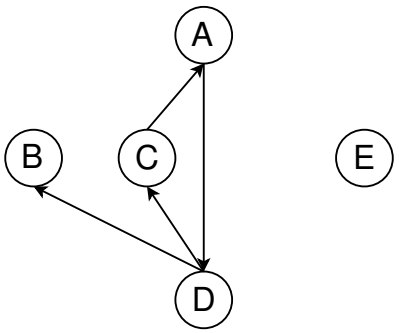
We said that we would separate between *potential* and *actual* Walls, similar to DOTs and Cones in Sect. 5.2. In contrast to the Cones and Cone Trees the potential Walls will not be explicitly represented. Through connectedByWallPipe we have an indirect representation of them which will be used to check the requirements.

Definition 6.2 (Potential Walls). *A potential Wall is a connection component of the graph constituted by all spheres as the vertices and the Wall Pipe as the edges.*

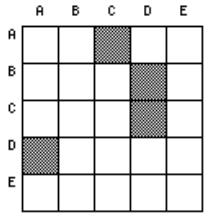
When two spheres are connected by a Wall Pipe they belong to the same (potential) Wall. The pipe connectivity is a necessary and sufficient criterion. If all pairs of spheres that are connected by Wall Pipes fullfill the structural and geometrical requirements, we can deduce that all potential Walls are actual Walls. It will not be necessary to access Walls as entities in their own right. In consequence, we do not have a direct representation of all spheres belonging to one Wall.

Why do we skip this step of explicitly representing Walls? For the Cone Trees it was important to have all Cones at hand in order to assemble them to Cone Trees and to compare the sizes of Cones belonging to the same Cone Tree (Cons. 5.10). The Walls do not have smaller non-atomic building blocks comparable to the Cones of a Cone Tree, and thus the information in connectedByWallPipe is sufficient.

Before studying the constraints, we must ensure that the Wall Pipes "function" correctly. The following two constraints are already known from



(a) A set of spheres connected by Wall Pipes.



(b) matrix representation of connectedByWallPipe for this example.

Figure 6.2: A set of interconnected spheres and their representation through connectedByWallPipe.

the Cone Pipes (Cons. 5.1 and Cons. 5.2). Again, we do not show their transformation from predicate logical statements into relation-algebraic ones, as it is analogous to the Cone variants.

Constraint 6.1. *No other entities than spheres can be connected to a Wall Pipe. Whenever a pipe p is a Wall Pipe it must be true that the element related to p via* pipeStarts *is a sphere as well as the element related to p via* pipeEnds*:*

$$\forall p\ \mathsf{wallPipe}_p \leftrightarrow \exists s\ \mathsf{pipeStarts}_{p,s} \wedge \mathsf{box}_s \wedge \exists t\ \mathsf{pipeEnds}_{p,t} \wedge \mathsf{sphere}_t$$
$$\Leftrightarrow \mathsf{wallPipe} = \mathsf{pipeStarts}\ ;\mathsf{sphere} \cap \mathsf{pipeEnds}\ ;\mathsf{sphere}.$$

Constraint 6.2. *The spheres incident to a Wall Pipe must be* different *so no loops on a sphere are possible. To express that only different spheres can be connected by the same pipe we demand:*

connectedByWallPipe *must be irreflexive.*

6.4 Constraints for Information Walls

As with Cone Trees, we first discuss the structural constraints followed by the geometrical constraints.

6.4.1 Structural Constraints

Information Walls represent inheritance structures for interfaces. Obviously we want no sphere to participate in two different representations of inheritance structures, and therefore in two different Walls.

Constraint 6.3. *Every sphere is part of at most one Wall.*

Constraint 6.3 is implied: To appear in two Walls at the same time a sphere would need to be connected to *two* different spheres by a Wall Pipe, t and u, that belong to different Walls. Since t is connected to s, they belong to the same Wall and since s is also connected to u they are part of the same Wall. Consequently, t and u are part of the same Wall, opposing our initial assumption. Hence, a Cone cannot belong to more than one Wall.

We have already shown how to see Walls as a graph with the spheres as vertices and pipes as edges. A *path* in a graph is a sequence of vertices that are connected by edges. A path x_0, x_1, x_2 implies that x_0 and x_1 are connected by an edge as well as x_1 and x_2. A cycle is a path where the first and the last path element are identical. In terms of our small example, we would have found a cycle if $x_0 = x_2$ would hold. The transitive closure of a graph (and thereby a relation) contains all nodes that are related by a path of any length greater than zero. If we find any path that connects a node with itself we found a cycle.

Constraint 6.4. *No cyclic connections of Wall Pipes can appear in a Wall; that is, no path of Wall Pipes may exists that connects a sphere with itself. The transitive closure of* connectedByWallPipe *must be irreflexive, which is expressed as:*

$$\mathsf{connectedByWallPipe}^{+} \subseteq \bar{\mathbf{I}}.$$

This constraint is a generalization of Constraint 6.2, since cycles of length one are cycles in a *path* of length one. The mathematical relatedness shows well when comparing the two constraints. We did not run into this redundancy with Cone Trees, as we excluded cyclic paths by showing that the geometrical constraints would not allow cycles (cf. Sect. 5.3). Therefore, Fig. 6.2 on the preceding page can not be a valid Wall as there is a cycle: (A, C, D, A).

6.4.2 Geometrical Constraints

Only one geometrical constraint needs to be introduced: all spheres of the same Wall must be arranged on one of the three planes parallel to the coordinate system. We can not say which plane that is and it is also completely valid that different Walls might have a different orientation. With all these additional requirements we can not expect to be able to find a "simple" constraint.

Our approach includes three steps: first we introduce a relation to accumulate all spheres of one Wall (similar to sameTree), then we define a group of relations that tells us which Walls have all spheres on one level (height, depth, width) and finally we can put the pieces together by defining a constraint which states that all spheres of one Wall must be aligned on one of the three levels. First, we introduce the relation that relates all spheres of the same Wall:

Relation sameWall : ***SPHERE*** $\leftrightarrow$ ***SPHERE*.** *This relation contains all pairs of spheres that belong to the same wall. Spheres belong to the same Wall if they can be reached through Wall Pipes. We need to abstract from the directions of the pipes and use the transitive closure of* $\mathsf{connectedByWallPipe} \cup \mathsf{connectedByWallPipe}^{\mathsf{T}}$*:*

$$\mathsf{sameWall}_{s,t} :\Leftrightarrow (\mathsf{connectedByWallPipe} \cup \mathsf{connectedByWallPipe}^{\mathsf{T}})^{+}_{s,t}.$$

The next step is to define relations that indicate if all spheres of one Wall are on the same level (same height, same depth or same width). We introduce a parametrized relation that works the same for all three coordinates:

Relation allSameLevel(sameLevel) : ***SPHERE*** $\leftrightarrow$ ***1*.** *This vector models the subset of spheres that are on the same level, whereas "same level" can be one of "same height", "same width", or "same depth". For a sphere* s *we can access all spheres* t *that are part of the same Wall by* sameWall *if* s *is part of a Wall. For any* t *must be true that it is on the same level as* s*:*

$$\begin{aligned}
&\mathsf{allSameLevel(sameLevel)}_s \\
&:\Leftrightarrow \mathsf{dom(connectedByWallPipe)}_s \wedge \forall t\ \mathsf{sameWall}_{s,t} \rightarrow \mathsf{sameLevel}_{s,t} \\
&\Leftrightarrow \mathsf{dom(connectedByWallPipe)}_s \wedge \mathsf{(sameWall/sameLevel)}_{s,s} && \text{(T8)} \\
&\Leftrightarrow \mathsf{dom(connectedByWallPipe)}_s \wedge \big(\mathsf{(sameWall/sameLevel)} ; \mathsf{L}\big)_s && \text{(T14)} \\
&\Leftrightarrow \big(\mathsf{dom(connectedByWallPipe)} \cap \mathsf{(sameWall/sameLevel)} ; \mathsf{L}\big)_s && \text{(T4).}
\end{aligned}$$

For one specific level, for example the *height*, this relation is a vector with those elements in it that are part of a Wall and where all spheres of that Wall are on the same height (x-coordinate). The two vectors for width and depth work accordingly. The important idea for the constraint is that we have to join all three vectors and check which elements are in the combined vector. If each Wall passes the "same level" criterion the vector should have each element in it that is participating in any Wall. If one Wall has its spheres neither on the same height, nor the same depth nor the same width, its elements are not part of the joined vector and we know that something is wrong:

Constraint 6.5. *If s is part of the domain or range of* connectedByWallPipe *it is part of a Wall. In this case s has to be either on the same height, the same width or the same depth as all the other spheres on that Wall:*

$$\begin{aligned}
&\forall s\ \mathsf{dom(connectedByWallPipe)} \vee \mathsf{ran(connectedByWallPipe)} \\
&\quad \rightarrow \mathsf{allSameLevel(sameHeight)}_s \vee \mathsf{allSameLevel(sameWidth)}_s \\
&\quad \vee \mathsf{allSameLevel(sameDepth)}_s \\
&\Leftrightarrow \forall s\ (\mathsf{dom(connectedByWallPipe)} \cup \mathsf{ran(connectedByWallPipe)})_s \\
&\quad \rightarrow (\mathsf{allSameLevel(sameHeight)} \cup \mathsf{allSameLevel(sameWidth)} \\
&\quad \cup \mathsf{allSameLevel(sameDepth)})_s \qquad \text{(T5)} \\
&\Leftrightarrow \mathsf{dom(connectedByWallPipe)} \cup \mathsf{ran(connectedByWallPipe)} \\
&\quad \subseteq \mathsf{allSameLevel(sameHeight)} \cup \mathsf{allSameLevel(sameWidth)} \\
&\quad \cup \mathsf{allSameLevel(sameDepth)} \qquad \text{(T11).}
\end{aligned}$$

We are now able to answer the question of "when is a set of spheres a Wall?"

Definition 6.3 (Valid Information Walls). *A set of spheres is Wall if it is a potential Wall and Constraints 6.1 through 6.5 hold.*

6.5 Entities for Walls

In the "version" of $\mathrm{J^3D_L}$ described in this thesis, no entity is introduced that could wrap a Wall. As a consequence, we cannot rule out that two Walls are overlapping even if no spheres and Wall Pipes are overlapping. It is possible, for example, that one Wall parallel to the $x-y$ plane "intersects" with a Wall on the $y-z$ plane in a way that no sphere or pipe overlaps.

To exclude such overlapping, the thoughts made in the context of Cone Shells could be applied to Walls. An entity "Wall Shell" could be introduced and related to each Wall. The remarkable difference in comparison with Cone Shell would be that no unique identification between Walls and Wall Shells could be made. It is not possible to determine one sphere that represents the Wall as it was possible for Cone Trees, because the Walls are not trees, and hence no unique root element can be identified.

On the other hand, each sphere is part of at most one Wall, which was not true for boxes and Cones. Therefore a constraint could generally rule out any overlapping between any Wall Shells.

6.6 Semantical Interpretation

Information Walls show inheritance networks in $\mathrm{J^3D_L}$. The inheritance is indicated by the Wall Pipes. Therefore we want to ensure that the Wall

Pipes are only drawn when the corresponding interfaces are in an inheritance relationship:

Constraint 6.6. *Interface i inherits from interface j if and only if a Wall Pipe runs between the spheres s and t that represent i and j:*

$$\begin{aligned}\forall i, j\ \mathsf{inheritsInt}_{i,j} \leftrightarrow \exists s, t\ \mathsf{interfaceOf}(t) = i \wedge \mathsf{interfaceOf}(s) = j \\ \wedge\ \mathsf{connectedByWallPipe}_{s,t} \\ \Leftrightarrow \mathsf{inheritsInt} = \mathsf{interfaceOf}^{\mathsf{T}}\ ;\mathsf{connectedByWallPipe}^{\mathsf{T}}\ ;\mathsf{interfaceOf}.\end{aligned}$$

This constraint is basically the same as Cons. 5.18 for Cone Trees and the transformation is very similar. If the above constraint holds, the Walls are valid *and* proper (cf. Def. 6.1 on page 68).

Although Cone Trees and Information Walls have very different geometrical and structural requirements, their connection to the ASG works in the same way. In both cases, the fact that two Java elements of the same type are related in a certain way is expressed by connecting two graphical entities with a pipe. When extending $\mathrm{J^3D_L}$ to show other relations between classes, for example, this same scheme of relational formalization could be used.

6.7 Conclusion

What have we learned by formalizing Information Walls for the use in $\mathrm{J^3D_L}$? The Walls themselves were not that interesting, other than they were the second non-atomic entity after Cone Trees. This allowed us to observe what and how much of the methods developed for Cone Trees were reusable. Trying to make any generalized statement from only two case studies would be a daring undertaking, yet we can see first tendencies on the nature of certain formalizations in $\mathrm{J^3D_L}$ that we want to present here.

Potential vs. Actual

The first, and already mentioned, important commonness between Cone Trees and Information Walls is that they are non-atomic entities. In both cases we decided to include an intermediate step of identifying potential entities. The argument for this extra step was the same in both cases and seems to have validity for other situations: when identifying a non-atomic entity usually several requirements need to be met in order for an assembly of atomic entities to be such a compound one. Out of these requirements we were able to isolate a sufficient criterion for two atomic entities to belong to the same non-atomic entities: the connection by pipes. Whenever two spheres or boxes are connected by a special kind of pipe they have to belong to the same composed entity, whether Wall or Cone Tree. This is only possible, of course, if the type of pipe is *exclusive* to one kind of composed

entity. Then we can argue that two entities connected by a pipe must be part of the specific entity, and if they do not posses all properties, the composed entity is invalid.

Direct vs. Indirect Representation

At one point the approaches differed for Walls and Cone Trees. While it was essential to represent Cones explicitly; i.e., to find a representing entity for the whole Cone Tree (the root box), nothing comparable was done for Walls. The reason to not have such an representation for Walls is that we did not use the Walls as entities in any further relation. The Cones were mapped to g-Cones, and Cone Trees were assembled of them. The constraints of Walls can be checked completely by referring to relations such as connectedByWallPipe that only indirectly represent single Walls and they were not used in any further way.

This insight can be helpful in other situations of relation-algebraic modeling: one can start and try to describe all constraints in terms of the (easier to find) indirect representations. If one succeeds for all requirements, there is no need to find either a tricky representation of composed entities (as it was done for Cones) or to model them as sets using the "expensive" membership relation.

Graph Structures

On a more general level, we modeled the nodes and edges of graphs twice. Structurally both Cone Trees and Information Walls are special types of graphs, the first a tree, the latter a DAG. In the SRG both nodes (spheres and boxes) and the edges (pipes) appear as entities. The connectedBy-relations we abstracted from the pipes, allowing us to describe graphs compactly by a homogeneous relation. This is, of course, only possible when the pipes have no exotic geometrical properties but behave "nicely" as we assumed in Assumption 4.1. This should be true for the greater part of visual languages so that our approach of abstracting from the pipes even if they appear as separate entities can be useful.

Semantical Interpretation

Semantically we had the most congruence between the two visualization techniques. Connecting the scene to a semantic interpretation is done by mapping the pipe connectivity to the semantic relation and having another relation to map entities into their semantic counterparts. Figure 6.3 sketches this situation. Whenever two visual entities are connected by a pipe, we know that their semantic counterparts (classes, respectively, interfaces) have to be related by the appropriate semantical relation. When discussing application scenarios for $\mathrm{J^3D_L}$ we will return to this thought.

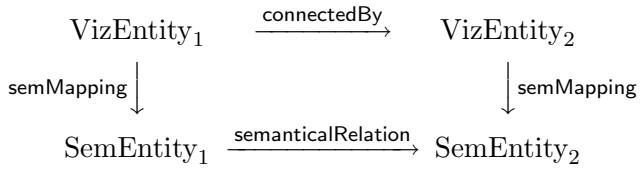


Figure 6.3: Schematic illustration of how the "pipe connectivity" in a 3D represents a semantical relation when having mappings between the visual and the semantical entities.

Now we turn to the last remaining visualization techniques that needs to be introduced: the Information Cube.

Chapter 7

Information Cubes

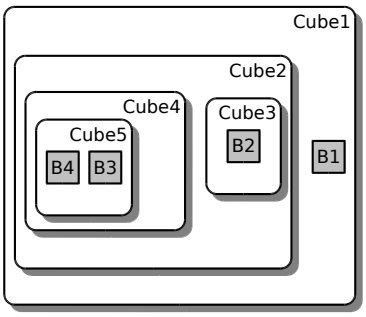
In J^3D_L the Cubes are used to display package membership of classes and interfaces. Packages can be nested, thereby forming a package hierarchy. and the classes and interfaces are the leaves of this hierarchy. The Cubes stand for the packages, boxes represent classes, and spheres represent interfaces as we already know. In the previous chapters we have described how spheres and boxes are arranged in Walls and Cone Trees. In this chapter we will leave aside the requirements of those visualization techniques and treat boxes and spheres as independent entities that are totally "free" and not integrated in any other structure. In Part III we will show how an integration of Cubes, Walls and Cone Trees can appear.

Section 7.1 develops the crucial relation with which we express that entities can be members of Cubes. Equipped with this relation, we can develop the structural constraints in Sect. 7.2 and the semantical issues in 7.2.2. In Sect. 7.3 we will sum up our discoveries.

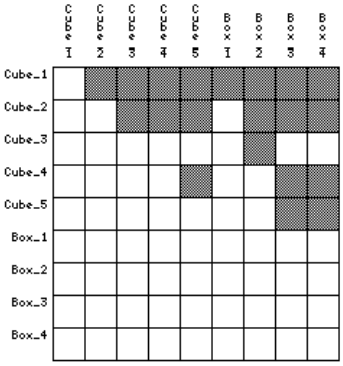
7.1 The Cube Membership Relation

The Cube, in contrast to the Cone Tree, *is* an atomic entity. There is no need to construct it from other entities; we can find Cubes directly in the SRG. The specialty of Cubes is that they are associated with other entities through *containment*. A box, an interface or a subcube "belongs" to a Cube if it is contained by it. Containment is the primary criterion for Cubes, just as the connection through Cone Pipes was for the Cone.

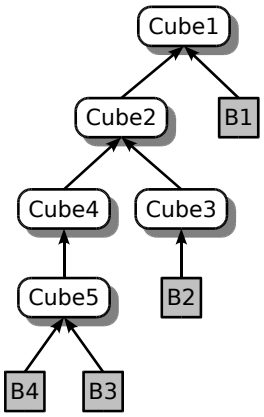
Figure 7.1 on the following page will serve as an example for the analysis regarding the adequacy of contains for our purposes. In Fig. 7.1(a) we can see a number of boxes and Cubes. If we read the diagram as a 2D projection of a 3D scene we can say that a box or Cube b is contained by a Cube c, when b is drawn inside of c. With this spatial infomation, a scene parser can construct the contains relation.



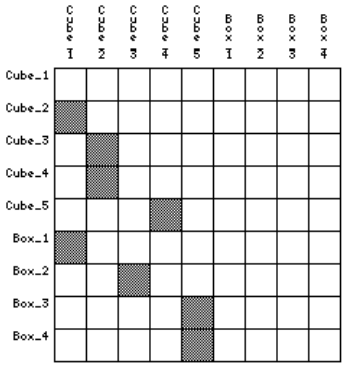
(a) Cubes (white rectangle with shadows) containing other Cubes and boxes (gray rectangles).



(b) The matching instance of contains as a matrix.



(c) The matching Cube/box membership diagram. Arrows indicate the *membership* relation.



(d) The matching instance of memberOf as a matrix.

Figure 7.1: An example scene of boxes contained by Cubes and its' depiction as a membership hierarchy.

When we introduced contains we demanded that it be irreflexive and transitive (Cons. 4.7). In itself nothing is "wrong" with the transitivity, but for our purposes it is a hindrance: when we want to say something such as "all boxes of the same Cone Tree must be inside the same Cube", contains gives us no direct answer on which boxes belong to the same Cube. Cube 1 of our example contains box 1 as well as box 4, but as we can see, box 1 and box 4 are in different Cubes. We need to develop a relation which reflects only the *direct containment* that relates Cube 1 with box 1 but not with box 4. With such a relation, we can determine the *inner-most* entity that contains another entity. To keep the concept of *direct containment* separate from the general containment, we introduce the term *member*.

Definition 7.1 (Membership of Entities). *An entity e is a member of an entity f if f contains e and e is not contained by any other entity g that is also contained by f.*

The following relation derives the membership out of contains according to the above definition.

Relation memberOf : $\boldsymbol{ENTITY \leftrightarrow ENTITY}$**.** *This relation contains all pairs of entities where the first is member of the second according to the definition of "membership" in Def. 7.1. Formally this translates into:*

$$\begin{aligned}
\mathsf{memberOf}_{e,f} : &\Leftrightarrow \mathsf{contains}^{\mathsf{T}}_{e,f} \wedge \neg\exists g\ \mathsf{contains}^{\mathsf{T}}_{e,g} \wedge \mathsf{contains}^{\mathsf{T}}_{g,f} \\
&\Leftrightarrow \mathsf{contains}^{\mathsf{T}}_{e,f} \wedge \overline{(\mathsf{contains}^{\mathsf{T}} ; \mathsf{contains}^{\mathsf{T}})}_{e,f} && (\mathsf{T7}) \\
&\Leftrightarrow \mathsf{contains}^{\mathsf{T}}_{e,f} \wedge \overline{(\mathsf{contains} ; \mathsf{contains})}^{\mathsf{T}}_{e,f} && (\mathsf{A6}) \\
&\Leftrightarrow \left(\mathsf{contains} \cap \overline{(\mathsf{contains} ; \mathsf{contains})}\right)^{\mathsf{T}}_{e,f} && (\mathsf{T4}).
\end{aligned}$$

It can be nicely seen that the relation takes the converse of contains and "cuts out" the transitive part (contains ; contains), which matches the definition of *membership*. In our example the instance of the contains relation of Fig. 7.1(b) leads to the instance of memberOf relation of Fig. 7.1(d). Only direct containment, the membership, is visible, visualized as the edges between the boxes, we result in seeing the membership *hierarchy* (Fig. 7.1(c)). Note that while we argued for boxes and Cone Trees only, the relation is phrased generally. The same thoughts are applicable for spheres as well. Conceptually, *membership* is "reverse" to *containment*: if entity a is a member of b this implies that b contains a. It is easy to see that $\mathsf{memberOf} \subseteq \mathsf{contains}^{\mathsf{T}}$ holds.

To further simplify the handling of Cube membership, we introduce a special case of the membership relation that considers Cubes as the only containing entities.

Relation cubeMemberOf : $\boldsymbol{CUBE \leftrightarrow ENTITY}$**.** *This relation contains all pairs of entities and cubes, where the first is member of the second. An*

entity e is a Cube member *of an entity c, if e is member of c and c is a Cube:*

$$\begin{aligned}
\mathsf{cubeMemberOf}_{e,c} : &\Leftrightarrow \mathsf{memberOf}_{e,c} \wedge \mathsf{cube}_c \\
&\Leftrightarrow \mathsf{memberOf}_{e,c} \wedge \mathbf{L}_e \wedge \mathsf{cube}_c^{\mathsf{T}} && (\mathsf{T14},\mathsf{T2}) \\
&\Leftrightarrow \mathsf{memberOf}_{e,c} \wedge (\mathbf{L} \,;\, \mathsf{cube}^{\mathsf{T}})_{e,c} && (\mathsf{T7}) \\
&\Leftrightarrow \left(\mathsf{memberOf} \cap \mathbf{L} \,;\, \mathsf{cube}^{\mathsf{T}}\right)_{e,c} && (\mathsf{T4}).
\end{aligned}$$

These concepts and their corresponding relations allow us to formalize the Cube constraints.

7.2 Cube Constraints

The requirements and constraints regarding the Cube as such are not very numerous or complicated. Later, the integration of Cone Trees and Information Walls will add more complexity.

The indication of "association through containment" is the crucial property of Cubes and membership gives us a tool to formalize this association. The general requirements for Cubes are:

- when a box, a sphere or a subcube overlap with a Cube they must be completely contained in the Cube,
- entities (boxes, spheres and Cubes) are only member of a Cube when they semantically belong to it.

7.2.1 Overlapping vs. Containment

As a visual clarification and for the sake of conceptual unambiguousness, we rule out certain cases of overlapping between boxes and Cubes. If a box or sphere overlaps with a Cube, the Cube must contain the box or sphere. The reason to demand this is clarity. The Cubes make use of the containment metaphor to indicate package membership. In this context a mere overlapping is ambiguous and confusing. Does a box that overlaps two-thirds with a Cube "belong" to it? What happens if a sphere is overlapping with two Cubes, that are nested themselves; does the sphere belong to the inner or the outer Cube? The membership relation does not cover these cases, since membership is derived from contains, and therefore only considers boxes and spheres that are already completely overlapping with a Cube.

Constraint 7.1. *When entity b is a box or a sphere and c is a Cube and both overlap, it must also be true that c and b are related via* contains*:*

$$\begin{aligned}
&\forall b, c\ (\mathsf{box}_b \vee \mathsf{sphere}_b) \wedge \mathsf{cube}_c \wedge \mathsf{overlap}_{b,c} \rightarrow \mathsf{contains}_{c,b} \\
&\Leftrightarrow \forall b, c\ \left((\mathsf{box} \cup \mathsf{sphere})\ ;\mathsf{cube}^{\mathsf{T}}\right)_{b,c} \wedge \mathsf{overlap}_{b,c} \rightarrow \mathsf{contains}_{c,b} && (\mathsf{T5},\mathsf{T7}) \\
&\Leftrightarrow \forall b, c\ \left((\mathsf{box} \cup \mathsf{sphere})\ ;\mathsf{cube}^{\mathsf{T}} \cap \mathsf{overlap}\right)_{b,c} \rightarrow \mathsf{contains}^{\mathsf{T}}_{b,c} && (\mathsf{T4},\mathsf{T2}) \\
&\Leftrightarrow (\mathsf{box} \cup \mathsf{sphere})\ ;\mathsf{cube}^{\mathsf{T}} \cap \mathsf{overlap} \subseteq \mathsf{contains}^{\mathsf{T}} && (\mathsf{T11}).
\end{aligned}$$

The same discussion applies for the containment of Cubes within Cubes. It is a desired feature that Cubes can be nested in other Cubes, but it must be true that the containment is complete. In this case, we cannot determine which entity must be contained inside which. With boxes and spheres it is clear that the Cube is the entity that contains and the box or sphere the entity that is contained inside the Cube. For two Cubes we can only demand that one of the two must be inside the other.

Constraint 7.2. *When b and c are Cubes and related via* overlaps *this must imply that they are also related via* contains *or* $\mathsf{contains}^{\mathsf{T}}$*:*

$$\begin{aligned}
&\forall b, c\ \mathsf{cube}_b \wedge \mathsf{cube}_c \wedge \mathsf{overlap}_{b,c} \rightarrow \mathsf{contains}_{b,c} \vee \mathsf{contains}_{c,b} \\
&\Leftrightarrow \forall b, c\ (\mathsf{cube}\ ;\mathsf{cube}^{\mathsf{T}})_{b,c} \wedge \mathsf{overlap}_{b,c} \rightarrow \mathsf{contains}_{b,c} \vee \mathsf{contains}^{\mathsf{T}}_{b,c} && (\mathsf{T2},\mathsf{T7}) \\
&\Leftrightarrow \forall b, c\ (\mathsf{cube}\ ;\mathsf{cube}^{\mathsf{T}} \cap \mathsf{overlap})_{b,c} \rightarrow (\mathsf{contains} \cup \mathsf{contains}^{\mathsf{T}})_{b,c} && (\mathsf{T4},\mathsf{T5}) \\
&\Leftrightarrow \mathsf{cube}\ ;\mathsf{cube}^{\mathsf{T}} \cap \mathsf{overlap} \subseteq \mathsf{contains} \cup \mathsf{contains}^{\mathsf{T}} && (\mathsf{T11}).
\end{aligned}$$

7.2.2 Semantical Constraints

The semantical constraints for Cubes can only be developed preliminarily at this time. Before we have discussed the integration of Cone Trees and Information Walls with Cubes, we cannot give an adequate description of the necessary requirements. Therefore, we will give a version of constraints, assuming that boxes and spheres are directly integrated as if they would not be part of Cone Trees or Walls. This way we can show how a simple integration of Cubes, spheres and boxes would look like but are able to also keep in mind that later on the integration of Cone Trees and Walls "breaks" the more simple assumptions. Willfully ignoring the constraints of Cone Trees and Walls we would demand:

Constraint 7.3. *A class, interface, or package c is member of a package p exactly if a box, sphere, or Cube b exists, which is part of the Cube k that*

represents the package:

$$\begin{aligned}
&\forall c, p\ \mathsf{packageMemberOf}_{c,p} \leftrightarrow \exists b, k\ \mathsf{cubeMemberOf}_{b,k} \\
&\quad \wedge (\mathsf{classOf}(b) = c \vee \mathsf{interfaceOf}(b) = c \vee \mathsf{packageOf}(b) = c) \\
&\quad \wedge \mathsf{packageOf}(k) = p \\
&\Leftrightarrow \forall c, p\ \mathsf{packageMemberOf}_{c,p} \leftrightarrow \exists b, k\ \mathsf{cubeMemberOf}_{b,k} \\
&\quad \wedge (\mathsf{classOf}_{b,c} \vee \mathsf{interfaceOf}_{b,c} \vee \mathsf{packageOf}_{b,c}) \\
&\quad \wedge \mathsf{packageOf}_{k,p} \qquad \text{(T13)} \\
&\Leftrightarrow \forall c, p\ \mathsf{packageMemberOf}_{c,p} \leftrightarrow \exists b, k\ \mathsf{cubeMemberOf}_{b,k} \\
&\quad \wedge (\mathsf{classOf} \cup \mathsf{interfaceOf} \cup \mathsf{packageOf})^{\mathsf{T}}_{c,b} \\
&\quad \wedge \mathsf{packageOf}_{k,p} \qquad \text{(T5,T2)} \\
&\Leftrightarrow \forall c, p\ \mathsf{packageMemberOf}_{c,p} \leftrightarrow ((\mathsf{classOf} \cup \mathsf{interfaceOf} \\
&\quad \cup \mathsf{packageOf})^{\mathsf{T}} ; \mathsf{cubeMemberOf} ; \mathsf{packageOf})_{c,p} \qquad \text{(T7)} \\
&\Leftrightarrow \mathsf{packageMemberOf} = (\mathsf{classOf} \cup \mathsf{interfaceOf} \\
&\quad \cup \mathsf{packageOf})^{\mathsf{T}} ; \mathsf{cubeMemberOf} ; \mathsf{packageOf} \qquad \text{(T12).}
\end{aligned}$$

Later when we integrate the different visualization techniques, boxes and spheres can appear in packages even if they semantically do not belong there. It will still be necessary for all boxes and spheres to appear in the Cube they semantically belong to. Constraint 7.3 will be modified to be an implication instead of an equivalence. To understand *why* this modification is necessary we need to describe the details of this integration, which happens in the following part of the thesis.

7.3 Conclusion

The Information Cube is the only visualization technique of $\mathrm{J^3D_L}$ that is based on atomic entities. There was no need to describe the assemblation of Cubes out of any other entity and therefore we turned directly to the analysis of how they integrate with boxes and spheres and how that expresses package membership.

The crucial criterion for Information Cubes is that they utilize *containment* to show the connection between an entity and a Cube. Hence, we analyzed the concept of containment as given by contains and realized that it needed to be redefined in two aspects:

directness: to separate direct containment from indirect containment, we introduce membership, which allowed us to see only those pairs of entities which are directly contained via contains, i.e. the non-transitive cases of containment.

completeness: the completeness of containment was ensured by constraint. We argued that from a visual point of view it would be confusing to have some entities not being completely contained by Cubes but only overlapping. In such cases, it would not have been intuitively clear whether or not an entity would "belong" to a Cube or not. Therefore, we installed a constraint that rules out entities that overlap with a Cube but are not contained by it.

The semantical aspects of Information Cubes could only be discussed preliminary due to the fact that the direct integration of spheres and boxes is an intermediate step towards our goals. In the next chapters we will discuss the integration of Cone Trees and Information Walls with Cubes, which will bring up some requirements that we have not discussed yet. In the "simple" scenario that we discussed, each sphere and box has to appear in exactly the Cube it semantically belongs to. Although this is not sufficient for J^3D_L, it would be of use if, for example, a 3D version of a UML package diagram should be implemented.

We have now introduced every visualization technique that we wanted to cover. On the Java level we are now able to (separately) create diagrams for class inheritance, interface inheritance and package membership. The next step will be to integrate these independent views of the code.

Part III

Integrating the Language's Parts

Chapter 8

Integration of Cone Trees with Cubes

The focus of the previous chapters was put on understanding the individual visualization techniques, finding appropriate relations to describe their features, and developing appropriate constraints. The attempt of this thesis is not only to describe the techniques themselves, but to discuss their integration as well.

One of the proposed benefits of three-dimensional modeling compared to two-dimensional approaches is the possibility to display more information and even combine different views of information. In the case of J^3D_L, the integration brings together the inheritance and the package membership view - two separate types of diagrams in the UML. It is beyond the scope of this thesis to make any statements about the actual perceptual benefits of such an integration. For us, it is interesting to see how the integration is performed relation-algebraically and how possible conflicts between constraints of the individual techniques can be resolved. We have three axes of integration:

1. Cubes with Cone Trees.
2. Cubes with Information Walls.
3. Information Walls with Cone Trees.

As we will see, the first two show a great amount of similarity, as they are both assembled out of interconnected entities that represent inheritance structures. When integrating the different visualization techniques, we will have geometrical problems to solve but since the different visualization techniques stand for different views onto the underlying Java code there will be also semantical challenges. The last step of integration, combining Walls and Cone Trees, will leave us with mixed results: while it is possible to be

described with relational methods it shows that the integration of all three visualization techniques is problematic, especially with the chosen metaphor of the Cube.

This chapter is devoted to the first axis of integration, the Cone Tree-Cube integration, and it is structured as following: First we describe how exactly the integration will happen in Sect. 8.1. This description will show the necessity to discuss the notion of (spatially) complete containment of Cone Trees in Cubes, which is defined in Sect. 8.2. The initial discussion will also uncover the need for a new type of box, which we will introduce in Sect. 8.3. In Sect. 8.4 and 8.5 we will deal with two phenomenons that are the result of the integration: (semantical) correspondence between Cone Trees and the avoidance of unnecessary Cone Trees.

8.1 The Outline of the Cone Tree–Cube Integration

When we integrate Cone Trees and Cubes in the same scene we will be able to see class inheritance and package membership at the same time. While this might be beneficial for the user of a J^3D_L-capable editor, it forces us to carefully analyze the consequences this integration has for the existing constraints, and to understand which new requirements arise. In particular, two questions need to be answered:

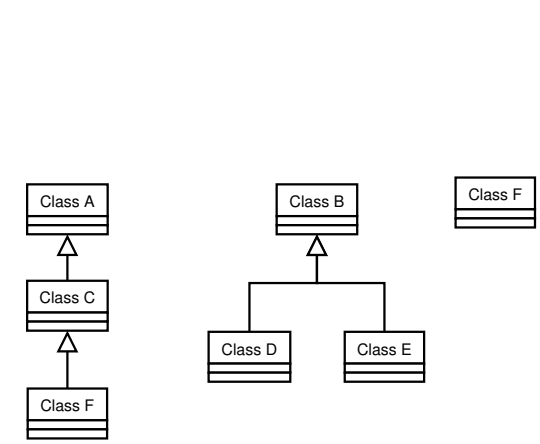
Question 1: on which level do the Cone Trees and Cubes integrate? Is there one Cube for each Cone Tree, or do Cone Trees span several Cubes?

Question 2: which semantical issues arise from the chosen integration? Is it possible to coherently show both inheritance and package membership at the same time, or are we forced to modify the visualization?

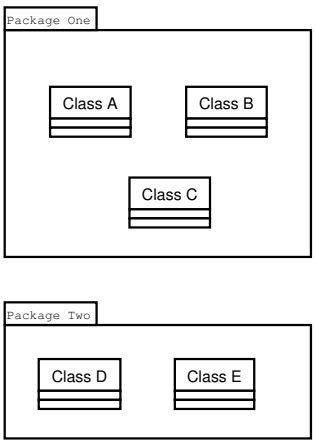
The short answer to these questions is: the complete Cone Trees will be shown in one Cube, and as a consequence, it is necessary to introduce a new type of box that indicates that its class does not belong to the package the box is located in. In the course of this chapter we make the following assumption:

Assumption 8.1. *A complete Cone Tree; that is, a Cone Tree with all of its boxes, is* supposed *to be rendered inside a Cube whenever at least one box of the Cone Tree* semantically belongs *to the Cube. A box belongs to a Cube semantically if the box's class is member of the Cube's package.*

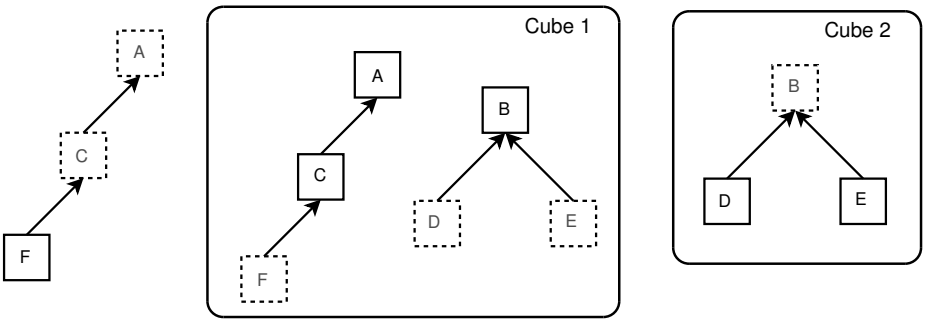
As an example, look at Fig. 8.1 on the next page. We see two UML diagrams: a class diagram with an inheritance structure (Fig. 8.1(a)) and a package diagram (Fig. 8.1(b)). The resulting J^3D_L structure is displayed



(a) The inheritance hierarchy of the classes.



(b) The package membership of the same classes.



(c) The resulting scene in J^3D_L. Dotted boxes indicate that the classes of the boxes are not part of the Cube's package.

Figure 8.1: An example sketching the intended way of integrating *inheritance* with *package membership*, and therefore Cone Trees with Cubes.

schematically in Fig. 8.1(c). In Cube 1, we see two Cone Trees and we see that A, B and C indigenously belong to the package of the Cube. Cube 2 has one Cone Tree with D and E as indigenous boxes. The Cone Tree to the far left is residing in no package, which can be interpreted as the *default package* in the context of Java, and we see that box F belongs to no package. Assumption 8.1 and the example we have just discussed help us to discover the consequences our approach has:

- Cone Trees reside inside Cubes completely,
- boxes do exist that reside in a Cube, although they do not belong there semantically (those with dotted lines in Fig 8.1(c)). This is the case for the boxes that represent classes which do not belong to the Cubes package,
- a Cube can contain multiple Cone Trees. Whenever several boxes belong to the Cube but are part of different Cone Trees, all Cone Trees will be shown inside the Cube,
- to show the package membership for every class, it must always appear exactly once inside the Cube it belongs to,
- for parsimony reasons, no Cone Tree must exist that is solely assembled out of boxes that do not belong into the package semantically.

Besides the integration of Cone Trees we have to also consider *single* boxes, which are not Cone Trees (cf. Def. 5.2). Most of the thoughts of the "direct" integration between boxes and Cubes made in Chapter 7 apply, but we will have to analyze what exactly changes and where a differentiation between boxes in Cone Trees and single boxes is necessary. The general requirements for single boxes are:

- single boxes must appear in the Cube they (semantically) belong to. This is not different from any other type of box,
- single boxes appear *exclusively* in the Cube they belong to. They are not connected to any other box that could belong to another Cube, hence there is no need for the box to appear in a Cube other than the one it belongs to. This is different to boxes that are connected to other boxes, and therefore part of a Cone.

8.2 "Complete" Containment of Cone Trees

How do we assert that a Cone Tree is completely (spatially speaking) inside a Cube? We will give three different answers to this question.

8.2.1 Three Alternative Approaches

The reason for such an elaborate discussion of this subject is twofold: we can show how the choice of each alternative has different consequences of the implementation in an editor, and it also allows us to illustrate our position regarding how much the visual language is supposed to predefine aspects of the usage of an implementation of the language.

Approach 1: "Box Containment". The first approach determines the containment of a Cone Tree solely by looking at the boxes of the Cone Tree: *A Cone Tree is completely contained by a Cube if all boxes of the Cone Tree are contained by the Cube.*

Approach 2: "Cone Shell Containment". The second approach incorporates the Cone Shell that we defined in Sect. 5 as the smallest g-Cone that fits around the pipes of a Cone, plus the cubical bulges for the parent and child boxes: *A Cone Tree is contained by a Cube if the Cone Shells of all Cones of the Cone Tree are contained by the Cube.*

Approach 3: "Cone Tree Shell Containment". To motivate the next definition, we must introduce the *rotation* of Cone Trees briefly mentioned in Chapter 5. A Cone of a Cone Tree can be rotated around the center of its vertex. The relative position of vertex and base boxes remains the same, so that all Cones connected to base boxes are also rotated. We also introduce here a hypothetical *Cone Tree Shell* which is defined similar to the Cone Shell of Sect. 5.4.

Definition 8.1 (Cone Tree Shell). *The shell of a Cone Tree is the smallest set of points that includes all Cone Shells in any possible rotation.*

With the Cone Tree Shell the third approach is described by: *A Cone Tree is contained by a Cube if its Cone Tree Shell is contained by the Cube.*

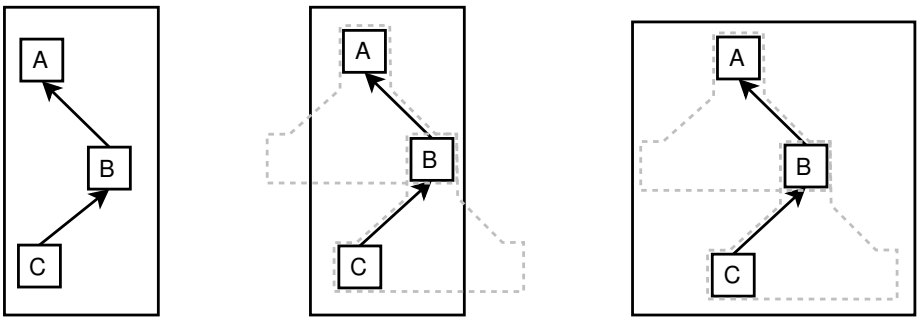
8.2.2 Choosing an Approach

We can see that the later approaches include the earlier. A Cone Tree with all Cone Shells inside a Cube will always include all boxes, because the boxes reside inside the Cone Shell. In the same way, a Cone Tree Shell that is inside a Cube will necessarily include all Cone Shells, and therefore implies that all Cone Shells of that Cone Tree are inside the Cube.

What properties do the different alternatives expose?

Box Approach vs. Cone Shell Approach

Fig. 8.2 on the following page compares Approach 1 with Approach 2. In Fig. 8.2(a) we see a Cone Tree inside the Cube according to Approach 1. As



(a) A Cone Tree validly contained by a Cube according to Approach 1.

(b) The same Cone Tree and Cube, showing how the Cone Shells would protrude from the Cube.

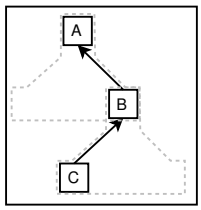
(c) The same Cone Tree validly contained inside according to Approach 2 in a resized Cube

Figure 8.2: An example illustrating the box approach of complete containment vs. the Cone Shell approach.

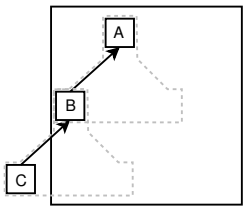
Fig. 8.2(b) shows, the Cone Shells would protrude from the Cube. This is problematic, as the Cone is intended to be rendered with its g-Cone showing, surrounding the pipes as in Screenshot 5.1 on page 40. Approach 2 does not allow the g-Cone to protrude, as the Cone Shell includes the g-Cone. This is shown in Fig. 8.3(a). So far, Approach 2 is preferable over Approach 1.

Cone Shell Approach vs. Cone Tree Shell Approach

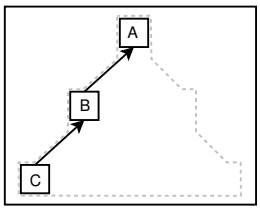
As we have seen, rotation is an essential operation performed on Cone Trees by the user. To look at Fig. 8.3 on the following page, we see the Cone from the above example rotated by the upper Cone. As a consequence, the lower Cone was also rotated and was moved outside the Cube. One could argue that the language should respect this operation, and that it should (in our case) enforce that rotation in a Cube is possible without ever violating any constraint. Such a standpoint would be covered by Approach 3, as it embeds the complete Cone Tree in a shell that is designed to cover any possible rotation (see Def. 8.1). Fig. 8.3(c) on the next page shows how the Cone Tree with the Cone Tree Shell is safely contained inside the Cube and that no rotation can change this. Of course, Approach 3 consumes even more space than the earlier ones. The space consumption might become a problem when the Cone Trees become large. Imagine a deep Cone Tree with no siblings on any level (a *chain*). The users sees a chain of boxes but is



(a) The Cone Tree of Fig. 8.2, validly contained in the Cube according to Approach 2.



(b) The same Cone Tree rotated by the upper Cone.



(c) The same Cone Tree wrapped by the "Cone Tree Shell" and validly contained by the Cube according to Approach 3.

Figure 8.3: Continuing the example of Fig. 8.2, comparing Cone Shell vs. Cone Tree Shell approach.

not possible to bring it any closer than to the Cubes, as as "safety distance" must be kept in order to ensure rotation.

Choosing One Approach

The question is: which of the three alternatives is the appropriate one? Approach 1 is plainly not sufficient, Approach 2 is not "rotation safe" and Approach 3 demands a lot of space for the presentation of a Cone.

We believe that Approach 2, the Cone Shell approach, is the appropriate one to choose. *Rotation* is a concept in the realm of interaction with a Cone Tree. When a scene is checked against the $\mathrm{J^3D_L}$ constraints it is static. It is a snapshot of the possible interactive environment that the user sees. When a scene is checked the positions of the Cones of a Cone Tree are fixed. The language should not have to consider interactivity to a degree where it becomes disadvantageous (the space consumption of Approach 3). If the user wants to rotate a Cone, the user or the editor have to take care that the Cube size and position is appropriate after rotation.

Therefore we can state Approach 2 as a definition:

Definition 8.2 (Complete Containment of Cone Trees). *A Cone Tree is contained by a Cube if the Cone Shells of all Cones of the Cone Tree are contained by the Cube.*

The consequence of this definition is that it is possible that the user rotates a Cone Tree so that parts of it protrude. If the user does nothing to fix such a situation, a yet-to-be-defined constraint will be violated when the scene is validated the next time, and an error will be produced.

8.2.3 Formalizing the Chosen Approach

Having chosen the Approach 2, we will now develop the constraints for the approach. We want to ensure that if a Cone Shell is overlapping with a Cube it is completely inside the Cube.

This type of requirement seems familiar; in fact, we had introduced similar requirements for Cubes, boxes and spheres in Sect. 7.2. Note that it includes no statement concerning whether or not the Cone Shell is "rightfully located" in the Cube (semantically speaking), as Cone Shells have no semantical interpretation. They are merely "middle-men" between boxes and Cubes. When a box appears in a Cube we know that part of the Cone Shell will be overlapping with the Cube, and together with the above requirement, we know that the complete Cone Shell will be contained by the Cube. The semantical issue of "package membership" is still one between boxes and Cubes, and Constraint 7.3 is still in effect to enforce the semantically correct display of boxes in Cubes.

Due to the similarity of this requirement to Constraint 7.1, we extend the constraint to include Cone Shells. The transformation remains the same.

Constraint 8.1 (Extension of Cons. 7.1). *When entity b is a box, a sphere or a Cone Shell and c is a Cube and both overlap, it must also be true that c and b are related via* contains*:*

$$\forall b, c\ (\mathsf{box}_b \vee \mathsf{sphere}_b \vee \mathsf{coneShell}_b) \wedge \mathsf{cube}_c \wedge \mathsf{overlap}_{b,c} \rightarrow \mathsf{contains}_{c,b}$$
$$\Leftrightarrow (\mathsf{box} \cup \mathsf{sphere} \cup \mathsf{coneShell})\ ;\mathsf{cube}^{\mathsf{T}} \cap \mathsf{overlap} \subseteq \mathsf{contains}^{\mathsf{T}}.$$

The above requirement argues about individual Cones Shells and therefore individual Cones, but our goal is to have complete Cone Trees inside a single Cube. The obvious question which arises is if this requirement is sufficient for our goal? The union of the Cone Shells of the same Cone Tree form a continuous surface. If a Cone Tree with more than one Cone would have Cones in two different Cubes, one Cone must exist that "leaves" one Cube in order to "reach" the next Cube. This would conflict with our requirement, since it would overlap with the first Cube but would not be contained by it. Therefore, Cons. 8.1 is sufficient.

While the handling of Cone Shell containment was straightforward until this point, we have to consider the consequences of introducing Cone Shells as entities. In Sect. 7.1, we introduced memberOf as a direct version of containment. Boxes and spheres were related to the *inner-most* Cube that contained them. According to this relation (and its slight variation cubeMemberOf), a box in a Cone Shell cannot be a member of a Cube: When a box is part of a Cone Tree it is contained by a Cone Shell and if the Cone Tree is inside a Cube, the Cube contains the Cone Shell and the box. In terms of memberOf the box is not member of the Cube as the Cone Shell is an intermediate containing entity.

Fig. 8.4 on the following page serves as an example. It shows a Cube containing a Cone Shell with two boxes and a solitary box. The matching instance of the member relation is shown in Fig. 8.4(b) where boxes A and B are not members of the Cube.

We argued earlier that the Cone Shell is semantically meaningless, and therefore we want to make it "invisible" for the membership considerations. To achieve this we introduce an "extended" membership relation, which relates a box with a Cube when either the box is already a member of the Cube or the box is part of a Cone Shell that is part of the Cube.

Relation extendedCubeMemberOf : ***BOX*** $\leftrightarrow$ ***CUBE.*** *This relation "extends" the* cubeMemberOf *relation: box b is an extended member of Cube c if it is a member of c (via* cubeMemberOf*) or if it is contained by a Cone Shell s that is member of c:*

$$
\begin{aligned}
&\mathsf{extendedCubeMemberOf}_{b,c} \\
&:\Leftrightarrow \mathsf{memberOf}_{b,c} \vee \exists s\ \mathsf{coneShell}_s \wedge \mathsf{memberOf}_{b,s} \wedge \mathsf{memberOf}_{s,c} \\
&\Leftrightarrow \mathsf{memberOf}_{b,c} \vee \exists s\ (\mathsf{L} ; \mathsf{coneShell}^{\mathsf{T}} \cap \mathsf{memberOf})_{b,s} \wedge \mathsf{memberOf}_{s,c} \quad (\mathsf{T14},\mathsf{T4}) \\
&\Leftrightarrow \mathsf{memberOf}_{b,c} \vee \big((\mathsf{L} ; \mathsf{coneShell}^{\mathsf{T}} \cap \mathsf{memberOf}) ; \mathsf{memberOf}\big)_{b,c} \quad (\mathsf{T7}) \\
&\Leftrightarrow \big(\mathsf{memberOf} \cup (\mathsf{L} ; \mathsf{coneShell}^{\mathsf{T}} \cap \mathsf{memberOf}) ; \mathsf{memberOf}\big)_{b,c} \quad (\mathsf{T5}).
\end{aligned}
$$

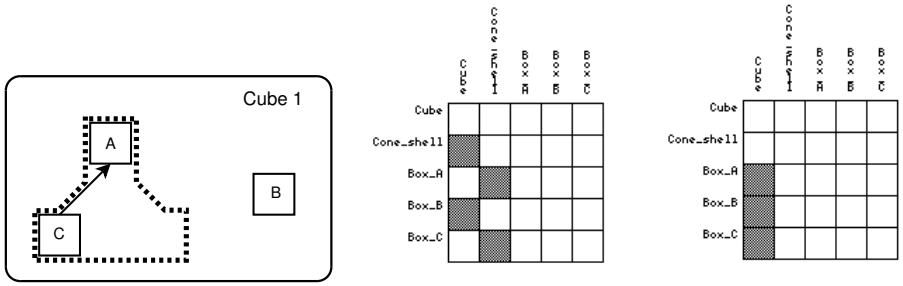
To better illustrate extendedCubeMemberOf, we show the matching instance of our small example. Fig. 8.4(c) shows that all three boxes are members of the Cube (in the extended notion), and that we only see membership of the Cubes; the Cone Shells do not appear. With this "filtered" relation, we will be able to handle the semantical properties as if Cone Shells would never have existed, while at the same time we can make use of the Cone Shells to formalize our concept of complete containment.

8.3 Ghost Boxes

Until now we have described in which way to completely contain a Cone Tree inside a Cube. By applying this containment policy, we end up with boxes that are inside a Cube although they do not belong there (cf. Fig. 8.1). The important things now are to describe a way to cope with these boxes relation-algebraically, and to discuss how far the fact that the same class can be represented by several boxes has consequences for our understanding of Cone Trees developed so far.

We will refer to such boxes as *ghost boxes*, as it is intended to display them "ghosted out" to indicate that they do not actually "belong" at their current location.

Assumption 8.2 (Ghost boxes vs. solid boxes). *We separate two kinds of boxes depending of their intended usage:*



(a) An example scene with one Cube containing a Cone Tree with its Cone Shell and a single box.

(b) The matching instance of member for this example.

(c) The matching instance of extended-CubeMemberOf for this example

Figure 8.4: A membership example illustrating the problem arising from the introduction of Cone Shells and it's solution.

- ghost boxes *are a special type of boxes that are intended to represent classes that are not members of the packages of the Cubes that they are located in,*
- solid boxes *are intended to represent classes that* do *belong into the packages that the Cubes represent they are located in.*

A Cone Tree that completely consists of ghost boxes is called a *Ghost Tree*. The existence of Ghost Trees is imaginable from a conceptual point of view. An editor would probably not allow the user to construct one, but that is not on the level we can argue in the language design.

8.3.1 Relations for Ghost Boxes

How do we know that some boxes in a Cube do not belong there without having a given packageOf relation? We expect the editor or the user to specify which boxes have the ghost box quality and specify which solid box is meant to be represented by the ghost. The task of J^3D_L is to verify that the ghost and solid boxes are used correctly. We have to introduce a new type of box and we will also have to modify some of the relations that deal with boxes. From now on we assume that the old set BOX is actually constructed by two *disjoint* subsets:

$$BOX = SOLID\text{-}BOX \cup GHOST\text{-}BOX.$$

For these we introduce the vectors:

Relation solidBox : ***ENTITY*** $\leftrightarrow$ ***1.*** *This vector represents the subset of entities that are solid boxes:*

$\mathsf{solidBox}_b$ holds if b is a solid box.

Relation ghostBox : ***ENTITY*** $\leftrightarrow$ ***1.*** *This vector represents the subset of entities that are ghost boxes:*

$\mathsf{ghostBox}_b$ holds if b is a ghost box.

As said above, we also expect the user or editor respectively to equip us with the information as to which ghost box is representing which solid box. For this purpose we assume a relation as the following:

Relation ghostBoxOf : ***GHOST-BOX*** $\leftrightarrow$ ***SOLID-BOX.*** *This relation contains all pairs of ghost boxes and solid boxes where the ghost box represents the solid box:*

$\mathsf{ghostBoxOf}_{b,c}$ holds if (ghost) box b represents the same class as (solid) box c.

Without the information of ghostBoxOf, it would not be possible for us to establish any correspondence between two Cone Trees that are supposed to represent the same inheritance structure in different Cubes (the intention behind the whole Cone Tree – Cube integration). By the following constraint we specify the expectations we have for the relation between ghost and solid boxes:

Constraint 8.2. *Each ghost box must represent exactly one solid box:*

ghostBoxOf *must be total and univalent.*

This constraint sounds trivial, but without it it would be possible to have ghost boxes that do not represent any existing solid box. This would give a whole new meaning to the term "ghost" (as in "ghost ship") but is something we definitely do not want to allow. Additionally, the constraint ensures that the ghost box does not stand for more than one solid box which would undermine its representational function. On the other hand, we of course still assume that more than one ghost box can represent the same solid box as part of different Cone Trees in different Cubes, hence the constraint makes no assumption about injectiveness.

8.3.2 Constraints for Ghost Boxes

Having introduced the ghost boxes and auxiliary relations, we have to ask ourselves: what does change by the introduction of ghost boxes? Changes might affect two aspects:

The individual Cone Tree does not change. There might be more than one Cone Tree in a scene representing the same inheritance structure, but for each Cone Tree taken for itself everything remains the same. A ghost box has all the geometrical properties that it's solid counterpart has, and it also represents the same class. All constraints for the individual Cone Trees are "unaware" of the two different kinds of boxes, at least they do not separate between them and can therefore remain unchanged.

The relationship between classes and boxes in general changes profoundly. In Sect. 4.4, we found that classOf is a *bijection* between boxes and classes. Exactly one box needs to exists to represent one class and each class is only represented once. With the introduction of ghost boxes this is of course, not possible any longer. We introduced the ghost boxes for the exact purpose of allowing a class to be represented more than once in different Cubes.

In order to deal with the new situation we take two steps:

- We restrict the old classOf relation to contain only solid boxes,
- and we introduce a relation for the ghost box that determines the class it (indirectly) represents.

To restrict classOf to only solid boxes we simply change the signature from

$$\mathsf{classOf} : \mathit{BOX} \leftrightarrow \mathit{CLASS}$$

to

$$\mathsf{classOf} : \mathit{SOLID\text{-}BOX} \leftrightarrow \mathit{CLASS}.$$

That way we preserve the bijection that existed between the boxes and classes. Before we integrated Cone Trees and Cubes, every box was "solid" in terms of our definition and now the solid boxes are still the "primary" representations of a class, as exactly one class must exist for one solid box. We restate the requirement for classOf so we are able to differentiate if we talk about the bijection of the *old* or the *new* version of classOf. We skip to show the complete constraint again, because it did not change, merely the signature of the relation it comprises has changed.

Constraint 8.3 (Restating of Cons. 4.13 on page 37). *Each* solid *box must be related to exactly one class and vice versa.*

Another change we need to consider is how inheritsClass must be related to connectedByConePipe, since now connectedByConePipe connects solid and ghost boxes in any combination. In order to express the new relationship, we need to know which class is represented (indirectly) by a ghost box. This is easily possible since we know which solid box a ghost box is related to:

Relation ghostBoxClassOf : *GHOST-BOX* $\leftrightarrow$ *CLASS*. *This relation contains all pairs of ghost boxes and classes where the ghost box indirectly represents the class. A ghost box b is related to a class c if ghost box b represents solid box d and c is the class of d:*

$$\begin{aligned}\mathsf{ghostBoxClassOf}_{b,c} &: \Leftrightarrow \exists d\ \mathsf{ghostBoxOf}_{b,d} \wedge \mathsf{classOf}(d) = c \\ &\Leftrightarrow (\mathsf{ghostBoxOf} ; \mathsf{classOf})_{b,c} && (\mathsf{T13},\mathsf{T7}).\end{aligned}$$

Looking at Constraint 8.2 and 8.3 we would expect:

Constraint 8.4. *Each ghost box has to have exactly one class associated with it.*

As we know that ghostBoxClassOf is assembled out of ghostBoxOf and classOf, we should be able to prove that the constraint is always true rather than to verify it for each 3D scene.

Theorem 8.1. ghostBoxOf *is total and univalent, given that Constraints 8.2 and 8.3 hold.*

Proof. First we show that ghostBoxClassOf is univalent.

$$\begin{aligned}&\mathsf{ghostBoxClassOf}^{\mathsf{T}} ; \mathsf{ghostBoxClassOf} \\ &= (\mathsf{ghostBoxOf} ; \mathsf{classOf})^{\mathsf{T}} ;\ (\mathsf{ghostBoxOf} ; \mathsf{classOf}) && (\text{definition}) \\ &= \mathsf{classOf}^{\mathsf{T}} ; \underbrace{\mathsf{ghostBoxOf})^{\mathsf{T}} ; \mathsf{ghostBoxOf}}_{\subseteq \mathsf{I},\ (\text{Cons. 8.2})} ; \mathsf{classOf} \\ &\subseteq \mathsf{classOf}^{\mathsf{T}} ; \mathsf{I} ; \mathsf{classOf} \\ &= \mathsf{classOf}^{\mathsf{T}} ; \mathsf{classOf} \\ &\subseteq \mathsf{I} && (\text{Cons. 8.3}).\end{aligned}$$

Now we prove that ghostBoxClassOf is total.

$$\begin{aligned}&\mathsf{ghostBoxClassOf} ; \mathsf{ghostBoxClassOf}^{\mathsf{T}} \\ &= (\mathsf{ghostBoxOf} ; \mathsf{classOf}) ;\ (\mathsf{ghostBoxOf} ; \mathsf{classOf})^{\mathsf{T}} && (\text{definition}) \\ &= \mathsf{ghostBoxOf} ; \underbrace{\mathsf{classOf} ; \mathsf{classOf}^{\mathsf{T}}}_{\supseteq \mathsf{I},\ (\text{Cons. 8.3})} ; \mathsf{ghostBoxOf}^{\mathsf{T}} \\ &\supseteq \mathsf{ghostBoxOf} ; \mathsf{I} ; \mathsf{ghostBoxOf}^{\mathsf{T}} \\ &= \mathsf{ghostBoxOf} ; \mathsf{ghostBoxOf}^{\mathsf{T}} \\ &\supseteq \mathsf{I} && (\text{Cons. 8.2}).\end{aligned}$$

□

Knowing that each ghost box has a class uniquely associated with it through ghostBoxClassOf, we can return to our original question regarding inheritsClass. Although we know that a solid box is the "original" place of a class representation, there exists not a unique locus of the inherits information in the SRG. The reason is that the same inheritance structure is displayed by possibly several Cone Trees in different Cubes, and that in each Cone Tree a Cone Pipe exists which indicates the inheritance information for the two classes. The Cone Pipe can run between any combination of solid and ghost boxes.

We will need to modify Cons. 5.18, but we must first split it up into two new constraints: one deals with the fact that the inheritance between two classes must correspond with at least two boxes (ghost and/or solid boxes) in the scene being connected by a Cone Pipe (the correspondence between ASG and SRG), and the other must ensure that all pairs of boxes that represent the same classes have a Cone Pipe connection if one pair has it (the correspondence on the SRG level).

In order to conveniently state that either a ghost box indirectly represents a class or that a solid box directly does so we introduce:

Relation extendedClassOf : ***BOX*** $\leftrightarrow$ ***CLASS.*** *This relation "extends" the* classOf *relation: box b represents class c if either b is solid and directly represents c via* classOf *or if b is a ghost and represents c indirectly via* ghostBoxClassOf*:*

$$\begin{aligned} \mathsf{extendedClassOf}_{b,c} &: \Leftrightarrow \mathsf{classOf}_{b,c} \vee \mathsf{ghostBoxClassOf}_{b,c} \\ &\Leftrightarrow (\mathsf{classOf} \cup \mathsf{ghostBoxClassOf})_{b,c} \qquad (\mathsf{T5}). \end{aligned}$$

The relation is basically a union of two existing relations. It is easy to see that this relation is also total and univalent, so we pass on without a formal proof.

Constraint 8.5 (Modification of Cons. 5.18 on page 65). *If class c inherits class d, two boxes e and f must exist, that are related to the classes via* extendedClassOf *and that are connected by a Cone Pipe:*

$$\begin{aligned} &\forall c, d\ \mathsf{inheritsClass}_{c,d} \leftrightarrow \exists e, f\ \mathsf{extendedClassOf}_{e,c} \wedge \mathsf{extendedClassOf}_{f,d} \\ &\qquad \wedge \mathsf{connectedByConePipe}_{f,e} \\ &\Leftrightarrow \mathsf{inheritsClass} \\ &\qquad = \mathsf{extendedClassOf}^{\mathsf{T}} \,;\, \mathsf{connectedByConePipe}^{\mathsf{T}} \,;\, \mathsf{extendedClassOf}. \end{aligned}$$

The above constraint ensures that at least one pipe exists that connects two boxes representing the inheritance between two classes. It does not ensure all pairs of boxes that represent the same classes are connected by a pipe if one of them is. This is the problem of *correspondence* between certain Cone Trees, which deserves a more general treatment.

8.4 Correspondence Between Different Cone Trees

Now that we have integrated the ghost boxes on a general level, we can take a look at the greater picture. By allowing ghost boxes, we have potentially increased the number of Cone Trees in a scene; in addition to that, we now have Cone Trees that correspond with each other. Such Cone Trees represent the same inheritance hierarchy in different Cubes. Hence, our effort at this point must be that the number of additional Cone Trees is kept to a minimum and that the correspondence between Cone Trees is given in the correct way.

What can go wrong in terms of this correspondence? Fig. 8.5 on the next page gives a small example to illustrate the problem.

Although the boxes of the left Cone Tree represent the same three classes as the ones in the right Cone Tree and the distribution of ghost boxes and solid boxes is valid, we have a problem. The implied inheritance is different in both Cone Trees and would lead to problems once transformed into inheritsClass. Of course, other completely different situations can occur with the same basic problem. In order to deal with such situations we must identify if, and if so what, these situations have in common in order to forbid them.

The similarity of all such situations is that whenever two boxes exist that represent the same class but that are not part of a identical inheritance structure (as above), they will have parents which do not represent the same class. In our example box B has once A as a parent and once C. That alone is enough to identify that two Cone Trees should correspond because the classes they represent do not show the same inheritance structure. In any possible arrangement at least one box will differ in this way. In other words: if two boxes exist that represent the same class, *and* if one of them is a start or end for a Cone Pipe, the same must be true for the other box and the two boxes on the other end must also represent the same class.

In order to make use of our insight we need to formalize the "represents the same class" concept that was mentioned above. The boxes that represent the same class are all ghost boxes that are representing the same solid box and the solid box itself. We capture this as a relation:

Relation sameClass : $\boldsymbol{BOX \leftrightarrow BOX}$**.** *This relation contains all pairs of boxes (ghost and solid ones) that represent the same class. Two boxes b and c represent the same class if they are identical, if one of them is the ghost*

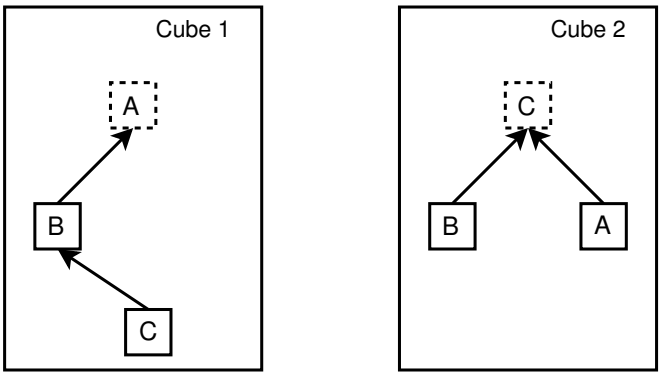


Figure 8.5: Invalid scene, where two Cone Trees with corresponding class names show two different inheritance structures. The Cone Tree in the Cube 1 represents the same classes as the Cone Tree in Cube 2, but has a different arrangement.

box of the other or if both are ghost boxes of the same solid box d:

$$\begin{aligned}
\mathsf{sameClass}_{b,c} : \Leftrightarrow\ & b = c \vee \mathsf{ghostBoxOf}_{b,c} \vee \mathsf{ghostBoxOf}_{c,b} \\
& \vee \exists d\ \mathsf{ghostBoxOf}_{b,d} \wedge \mathsf{ghostBoxOf}_{c,d} \\
\Leftrightarrow\ & (\mathsf{I} \cup \mathsf{ghostBoxOf} \cup \mathsf{ghostBoxOf}^{\mathsf{T}})_{b,c} \\
& \vee (\mathsf{ghostBoxOf} \,;\, \mathsf{ghostBoxOf}^{\mathsf{T}})_{b,c} \qquad (\mathsf{T5},\mathsf{T7}) \\
\Leftrightarrow\ & (\mathsf{I} \cup \mathsf{ghostBoxOf} \cup \mathsf{ghostBoxOf}^{\mathsf{T}} \\
& \cup \mathsf{ghostBoxOf} \,;\, \mathsf{ghostBoxOf}^{\mathsf{T}})_{b,c} \qquad (\mathsf{T5}).
\end{aligned}$$

Now it is easy to formalize the above as a constraint:

Constraint 8.6. *For each pair of boxes b and c must hold: if and only if a box x exists which represent the same class as b and is connected by a Cone Pipe to c there must exists another box y which is connected by a Cone Pipe to b and represents the same class as x:*

$$\begin{aligned}
& \forall b, c\ \exists x\ \mathsf{sameClass}_{b,x} \wedge \mathsf{connectedByConePipe}_{c,x} \\
& \quad \rightarrow \exists y\ \mathsf{connectedByConePipe}_{y,b} \wedge \mathsf{sameClass}_{y,c} \\
\Leftrightarrow\ & \forall b, c\ (\mathsf{sameClass} \,;\, \mathsf{connectedByConePipe}^{\mathsf{T}})_{b,c} \\
& \quad \rightarrow (\mathsf{connectedByConePipe}^{\mathsf{T}} \,;\, \mathsf{sameClass})_{b,c} \qquad (\mathsf{T7}) \\
\Leftrightarrow\ & \mathsf{sameClass} \,;\, \mathsf{connectedByConePipe}^{\mathsf{T}} \\
& \quad \subseteq \mathsf{connectedByConePipe}^{\mathsf{T}} \,;\, \mathsf{sameClass} \qquad (\mathsf{T11}).
\end{aligned}$$

To illustrate the Constraint we refer to Fig. 8.6 on the following page. It shows graphically what the scene needs to "look" like on the left and right

(a) The left side of the constraint

(b) The right side of the constraint

Figure 8.6: An illustration of Cons. 8.6.

side of the implication. We obtain a structure that reminds of a commutative diagram. In fact that we want to ensure a certain "commutativeness" in that both paths lead to the same result. Together with Cons. 8.5 on page 99, this ensures that the Cone Pipe is shown in all corresponding Cone Trees if appropriate (i.e., if an implement relation exists for the classes).

8.5 Avoiding the Redundant Display of Cone Trees

Finally we have to deal with a class of situations where unnecessary Cone Trees are displayed.

Two situations have to be ruled out: The one is that two Cone Trees represent the same inheritance hierarchy in the same Cube, and the other is that a Cone Tree is displayed in a Cube although none of its boxes belong to the Cube semantically (a Ghost Tree). Both are redundant.

Constraint 8.7. *If b and c represent the same class and are not identical, no Cube d can exist that both b and c are members of. For the sake of readability we substitute* extendedCubeMemberOf *with* exCMO*:*

$$
\begin{aligned}
&\forall b, c\ \mathsf{sameClass}_{b,c} \wedge b \neq c \rightarrow \neg\exists d\ \mathsf{exCMO}_{b,d} \wedge \mathsf{exCMO}_{c,d} \\
&\Leftrightarrow \forall b, c\ \mathsf{sameClass}_{b,c} \wedge \bar{\mathsf{I}}_{b,c} \rightarrow \neg(\mathsf{exCMO}\ ;\mathsf{exCMO}^{\mathsf{T}})_{b,c} && \text{(T3,T7)} \\
&\Leftrightarrow \forall b, c\ (\mathsf{sameClass} \cap \bar{\mathsf{I}})_{b,c} \rightarrow \overline{(\mathsf{exCMO}\ ;\mathsf{exCMO}^{\mathsf{T}})}_{b,c} && \text{(T4,T6)} \\
&\Leftrightarrow \mathsf{sameClass} \cap \bar{\mathsf{I}} \subseteq \overline{\mathsf{exCMO}\ ;\mathsf{exCMO}^{\mathsf{T}}} && \text{(T11).}
\end{aligned}
$$

If is sufficient to exclusively refer to boxes. If no two boxes can exist in the same Cube that represent the same class, no two Cone Trees can exist. Also we disallow that duplicate single boxes exist in one Cube which would have been possible for those boxes that are not part of a Cone Tree.
Next we turn to the Ghost Trees. We know that a Cone Tree is *not* a Ghost Tree if at least one box exists within it that is solid. We need to find a way to state a box that is part of a Cone Tree is connected to at least one

solid box directly or indirectly. Recall we said in Chapter 5, that a Cone is represented by its parent box, and that we could represent a complete Cone Tree by the parent of the top-most Cone in the Cone Tree. For this box, we can then reach every other box on the Cone Tree by the transitive closure of connectedByConePipe. This allows us to state the requirement without introducing any new relations.

Constraint 8.8. *Ghost Trees must not exist. Each box that is in the domain but not in the range of* connectedByConePipe *(these boxes represent the complete Cone Tree) must have a solid box related to it via the reflexive transitive closure of* connectedByConePipe*(It is important to use the reflexive transitive closure to cover the case where the top box itself is the only solid box). As before we abbreviate* connectedByConePipe *with* cbCP*:*

$$\begin{aligned}
&\forall b\ \mathsf{dom}(\mathsf{cbCP})_b \wedge \overline{\mathsf{ran}(\mathsf{cbCP})}_b \rightarrow \exists c\ (\mathsf{cbCP})^*_{b,c} \wedge \mathsf{solidBox}_c \\
&\Leftrightarrow \forall b\ (\mathsf{dom}(\mathsf{cbCP}) \cap \overline{\mathsf{ran}(\mathsf{cbCP})})_b \rightarrow (\mathsf{cbCP}^* \,;\mathsf{solidBox})_b \qquad (\mathsf{T4},\mathsf{T7}) \\
&\Leftrightarrow \mathsf{dom}(\mathsf{cbCP}) \cap \overline{\mathsf{ran}(\mathsf{cbCP})} \subseteq \mathsf{cbCP}^* \,;\mathsf{solidBox} \qquad (\mathsf{T11}).
\end{aligned}$$

With this constraint we have covered all possibilities of unnecessary Cone Trees. As before, we will discuss if the constraints so far cover all requirements of single boxes. The only constraint that needs to be restated for single boxes is the last one about Ghost Trees. Single boxes are not Cone Trees, and therefore not covered by the above constraint, but we still want to exclude single ghost boxes. A single ghost box is a "degenerated" Ghost Tree. It contains no "positive" information about package membership and should, therefore, be forbidden.

Constraint 8.9. *A single box is a box which is neither in the domain nor in the range of* connectedByConePipe*. Such a box must be a solid box. As before we abbreviate* connectedByConePipe *with* cbCP*:*

$$\begin{aligned}
&\forall b\ \overline{\mathsf{dom}(\mathsf{cbCP})}_b \wedge \overline{\mathsf{ran}(\mathsf{cbCP})}_b \rightarrow \mathsf{solidBox}_b \\
&\Leftrightarrow \forall b\ (\overline{\mathsf{dom}(\mathsf{cbCP})} \cap \overline{\mathsf{ran}(\mathsf{cbCP})})_b \rightarrow \mathsf{solidBox}_b \qquad (\mathsf{T4}) \\
&\Leftrightarrow \overline{\mathsf{dom}(\mathsf{cbCP})} \cap \overline{\mathsf{ran}(\mathsf{cbCP})} \subseteq \mathsf{solidBox} \qquad (\mathsf{T11}).
\end{aligned}$$

At this point we can conclude discussions about the ghost boxes and their requirements. We have achieved the goal of integrating Cone Trees with Cubes, and we have understood the consequences of this integration. Instead of summarizing the discussion now, we will look at the related discussion of Information Walls and Cubes. Afterwards we can give a comparative conclusion which is more interesting and informative.

Chapter 9

Integrating Walls with Cubes

The process of integrating Cubes with Walls will be similar to the one of Cubes and Cone Trees. As it will turn out, the integration this time is easier than the the other before. We will point out where the differences between the two lay and where it is similar. As before we will not show transformation of constraints and relations if they expose a similar structure to transformations we did in earlier chapters.

9.1 The Outline of the Wall–Cube Integration

Again, the questions are *how* does the integration work and *which* modifications are necessary; and again, the short answers are that complete Walls go in a Cube and we need to introduce a new type of sphere.

Assumption 9.1. *Information Walls are rendered completely inside a Cube if at least one sphere belongs there* semantically*. A sphere belongs to a Cube semantically if the interface of the sphere member of the Cube's package.*

The resulting list of properties is shown below. It is very similar to the list for Cone Trees and Cubes of Chapter 8:

- Walls reside inside Cubes. When a Wall is part of a Cube it has to be contained completely by the Cube,
- spheres exist that reside in a Cube, although they do not belong there semantically,
- a Cube can contain multiple Walls,
- to show the package membership for every interface, each sphere must appear in the Cube it belongs to exactly once, and
- for parsimony reasons, no Wall must exist that is solely assembled of spheres that do not belong into the Cube semantically.

Analogously, to the single boxes we have to consider the single spheres. As much as single boxes are not Cone Trees, the single spheres are not Walls. The demands for single spheres look familiar:

- single spheres must appear in the Cube they belong to. This is not different from any other type of sphere,
- single spheres appear only in the package they belong to. They are not connected to any other sphere that could belong to another Cube, hence there is no need for the sphere to appear in a Cube other than the one it belongs to.

9.2 Complete Containment of Walls

For Cone Trees we had a extended discussion about which notion of *complete containment* (spatially speaking) is the right one. The reason for the discussion was that each Cone is wrapped into a shell and the shell needed to be considered as well when defining the containment. For Walls, no such wrapping exists (cf. Sect. 6.5) and will define containment of Walls completely based on the spheres of the Wall, similar to Approach 1 on page 90 of the Cone Tree Containment discussion.

Definition 9.1. *A Wall is completely contained by a Cube, when all spheres of the Wall are members of the Cube.*

As a consequence, there is no need to use the extendedCubeMemberOf in order to formalize the Wall constraints, as the more general cubeMemberOf is suitable. When two spheres are inside a Cube, all points that lay on a straight line between the two points of the spheres are also inside the Cube. Therefore, the Wall Pipe that runs between two spheres is always contained by a Cube when the incident spheres are (cf. Assumption 4.1) and we do not need to deal with pipes separately.

The first constraint deals with the requirement that all Walls are either not part of a Cube or contained correctly according to Def. 9.1.

Constraint 9.1. *For all spheres s and Cubes c must be true: if another sphere t exists which is connected with s by a Cone Pipe (or the other way round) and if t is a cube member of c, sphere s is also a cube member of c:*

$$\begin{aligned}
&\forall s, c\ \exists t\ (\mathsf{connectedByWallPipe}_{s,t} \vee \mathsf{connectedByWallPipe}_{t,s}) \\
&\quad \wedge\ \mathsf{cubeMemberOf}_{t,c} \rightarrow \mathsf{cubeMemberOf}_{s,c} \\
&\Leftrightarrow \forall s, c\ \exists t\ (\mathsf{connectedByWallPipe} \cup \mathsf{connectedByWallPipe}^{\mathsf{T}})_{s,t} \\
&\quad \wedge\ \mathsf{cubeMemberOf}_{t,c} \rightarrow \mathsf{cubeMemberOf}_{s,c} && \text{(T2,T5)} \\
&\Leftrightarrow (\mathsf{connectedByWallPipe} \\
&\quad \cup\ \mathsf{connectedByWallPipe}^{\mathsf{T}})\ ;\mathsf{cubeMemberOf} \\
&\quad \subseteq\ \mathsf{cubeMemberOf} && \text{(T7,T12).}
\end{aligned}$$

Note that the containment of Walls does not conflict with any earlier constraints regarding Cube containment as was the case for boxes and Cone Trees. This is obviously due to the lack of any surrounding entity - Walls do not possess a wrapping shell. Individual spheres are still covered by Cons. 7.1, which states that the spheres overlapping with a Cube must be contained by the Cube. Since single spheres are not connected to any other spheres, being contained by a Cube is the most comprehensive form a containment.

9.3 "Ghost" Spheres

Being familiar with the discussion regarding ghost *boxes*, it comes as no surprise that Walls need ghost *spheres*. Walls can contain spheres that belong to several different packages. A sphere that appears in a Cube, although the interface it represents does not belong to the Cube's package, will be displayed "ghosted out". Any other sphere we will refer to as "solid". The process of developing the relations will be similar to ghost boxes with Cones and Cone Trees. In fact, the similarity is so close that often only the string "box" needs to be replaced with the string "sphere". The next definitions, relations, and constraints are introduced for analogous reasons their Cone Trees counterparts were introduced for. It is our goal to be complete without being redundant, so the argumentative prose is reduced to the minimum.

Assumption 9.2 (Ghost Spheres vs. Solid Spheres). *We separate two kinds of spheres depending of their intended usage:*

- ghost spheres *are a special type of spheres that are intended to represent interfaces that are not members of the packages of the Cubes that they are located in,*
- solid spheres *are intended to represent interfaces that* do *belong into the packages that the Cubes represent they are located in.*

9.3.1 Fundamental Relations for Ghost Spheres

First we introduce the necessary tools to deal with ghost spheres, which are: the sets, the vectors and the first requirements. As for boxes we redefine *SPHERE* as the disjoint union of two subsets:

$$SPHERE = SOLID\text{-}SPHERE \cup GHOST\text{-}SPHERE.$$

For these we introduce the vectors:

Relation solidSphere : ***ENTITY*** $\leftrightarrow$ ***1.*** *This vector models the subset of entities that are solid spheres:*

$\mathsf{solidSphere}_s$ holds if s is a solid sphere.

Relation ghostSphere : ***ENTITY*** $\leftrightarrow$ ***1.*** *This vector models the subset of entities that are ghost spheres:*

$\mathsf{ghostSphere}_s$ holds if s is a ghost sphere.

To connect ghost spheres with the solid spheres they represent we expect to have a relation available that contains these informations.

Relation ghostSphereOf : ***GHOST-SPHERE*** $\leftrightarrow$ ***SOLID-SPHERE.*** *This relation captures the relationship between the ghost spheres and the solid spheres they represent:*

$\mathsf{ghostSphereOf}_{s,t}$ holds if (ghost) sphere s represents (solid) sphere t.

Constraint 9.2. *Each ghost sphere must represent exactly one solid sphere. Hence we demand:*

ghostSphereOf *must be total and univalent.*

So far we have done nothing new nor exciting. We can now turn to look for potential changes of constraints and requirements regarding spheres and Walls.

9.3.2 Constraints for Ghost Spheres

As for the Cone Trees, we have to look for changes in two regards:

The individual Information Wall does not change. The original relations and constraints were not "aware" of the separation of spheres into solid ones and ghosts.

The relationship between interface and sphere in general changes in way that we are familiar with from classes and boxes. The solid spheres are the "true" representants of interfaces and we link the ghost spheres to interfaces by means of the new interfaceOf relation.

To restrict interfaceOf to only solid spheres we simply change the signature from

$$\mathsf{classOf} : \mathit{SPHERE} \leftrightarrow \mathit{INTERFACE}$$

to

$$\mathsf{classOf} : \mathit{SOLID\text{-}SPHERE} \leftrightarrow \mathit{INTERFACE}.$$

This way we again preserve the bijection between spheres and interfaces, only that we refined the spheres to be solid. Constraint 4.13 remains unchanged as the signature of the relation changed and not the asserted properties of the relation.

Constraint 9.3 (Restating of Cons. 4.13). *Each* solid *sphere must be related to one interface and vice versa.*

As before with ghost boxes we want to express the indirect relationship between ghost spheres and the interfaces they represent.

Relation ghostSphereInterfaceOf : ***GHOST-SPHERE*** $\leftrightarrow$ ***INTERFACE.*** *This relation contains all pairs of ghost spheres and interfaces where the ghost sphere is representing the interface. Ghost sphere s represents an interface i if s represents solid sphere t and i is the interface of t:*

$$\begin{aligned}\mathsf{ghostSphereInterfaceOf}_{s,i} &: \Leftrightarrow \exists t\ \mathsf{ghostSphereOf}_{s,t} \wedge \mathsf{interfaceOf}(t) = i \\ &\Leftrightarrow (\mathsf{ghostSphereOf}\,;\mathsf{interfaceOf})_{s,i}.\end{aligned}$$

Theorem 9.1. ghostSphereInterfaceOf *is total and univalent, assuming that Cons. 9.2 and Cons. 9.3 hold.*

The proof has the same structure as Theorem 8.1 on page 98, therefore it is not shown here.

9.4 Correspondence of Information Walls

With the ghost spheres comes a duplication of Walls, and therefore we have a correspondence between those Walls that represent the same inheritance structure. All has been analyzed for the Cone Trees, but as the inheritance of classes resembles a tree but the inheritance of interfaces resembles a DAG, we have to be examine if the solution for Cone Trees is applicable for Information Walls as well.

We again have two different constraints: the first states that for each inheritance between two interfaces at least one pair of spheres must exist that is connected by a Wall Pipe, and the other must ensure that all pairs of spheres that represent the same interface have a pipe if one of the pairs has a pipe. Before we turn to the first constraint, we must introduce a relation to "directly or indirectly" relate spheres to interfaces:

Relation extendedInterfaceOf : ***SPHERE*** $\leftrightarrow$ ***INTERFACE.*** *This relation "extends" the* interfaceOf *relation: sphere s is representing interface i if either it is a solid sphere and directly related to i or it is related to i indirectly being a ghost sphere:*

$$\begin{aligned}\mathsf{extendedInterfaceOf}_{s,i} : &\Leftrightarrow \mathsf{interfaceOf}_{s,i} \vee \mathsf{ghostSphereInterfaceOf}_{s,i} \\ &\Leftrightarrow (\mathsf{interfaceOf} \cup \mathsf{ghostSphereInterfaceOf})_{s,i}.\end{aligned}$$

With the above relation at hand, the constraint looks like this:

Constraint 9.4. *If interface i inherits interface j, two spheres s and t must exist that are related to the interfaces via* extendedInterfaceOf *and that are connected by a Wall Pipe:*

$$\begin{aligned}&\forall i, j\ \mathsf{inheritsInt}_{i,j} \leftrightarrow \exists s, t\ \mathsf{extendedInterfaceOf}_{s,i} \wedge \mathsf{extendedInterfaceOf}_{t,j} \\ &\quad \wedge \mathsf{connectedByWallPipe}_{t,s} \\ &\Leftrightarrow \mathsf{inheritsInt} \\ &\quad = \mathsf{extendedInterfaceOf}^{\mathsf{T}} \,;\, \mathsf{connectedByWallPipe}^{\mathsf{T}} \,;\, \mathsf{extendedInterfaceOf}.\end{aligned}$$

The problem of correspondence between Walls that represent the same inheritance structure is the same as the for Cone Trees; with the difference that the inheritance between interfaces has the structure of a DAG, not a tree. This difference is not relevant for our considerations, as we argued about individual pairs of boxes and their connection by pipes and we made no reference to the structure of the Cone Tree. Therefore it is no problem to take Cons. 8.6 on page 101 over. Before this we need to introduce sameInterface in analogy to sameClass:

Relation sameInterface : ***SPHERE*** $\leftrightarrow$ ***SPHERE.*** *This relation contains all pairs of spheres (ghost and solid ones) that represent the same interface. Two spheres s and t represent the same interface if they are identical, if one of them is the ghost sphere of the other or if both are ghost*

spheres of the same solid sphere u:

$$\begin{aligned}
\mathsf{sameInterface}_{s,t} : \Leftrightarrow\ & s = t \vee \mathsf{ghostSphereOf}_{s,t} \vee \mathsf{ghostSphereOf}_{t,s} \\
& \vee \exists u\ \mathsf{ghostSphereOf}_{s,u} \wedge \mathsf{ghostSphereOf}_{t,u} \\
\Leftrightarrow\ & (\mathsf{I} \cup \mathsf{ghostSphereOf} \cup \mathsf{ghostSphereOf}^{\mathsf{T}})_{s,t} \\
& \vee (\mathsf{ghostSphereOf} \,;\, \mathsf{ghostSphereOf}^{\mathsf{T}})_{s,t} \qquad (\mathsf{T5},\mathsf{T7}) \\
\Leftrightarrow\ & (\mathsf{I} \cup \mathsf{ghostSphereOf} \cup \mathsf{ghostSphereOf}^{\mathsf{T}} \\
& \cup \mathsf{ghostSphereOf} \,;\, \mathsf{ghostSphereOf}^{\mathsf{T}})_{s,t} \qquad (\mathsf{T5}).
\end{aligned}$$

With this relation, the constraint can be phrased:

Constraint 9.5. *For any two spheres s, and t must hold: if and only if a sphere x exists that represents the same interface as s and is connected to t by a Wall Pipe, a sphere y must exist that is connected to s and represent the same interface as t:*

$$\begin{aligned}
& \forall s, t\ \exists x\ \mathsf{sameInterface}_{s,x} \wedge \mathsf{connectedByWallPipe}_{t,x} \\
& \quad \leftrightarrow \exists y\ \mathsf{sameInterface}_{y,t} \wedge \mathsf{connectedByWallPipe}_{y,s} \\
\Leftrightarrow\ & \mathsf{sameInterface} \,;\, \mathsf{connectedByWallPipe}^{\mathsf{T}} \\
& = \mathsf{connectedByWallPipe}^{\mathsf{T}} \,;\, \mathsf{sameInterface}.
\end{aligned}$$

Except for the naming of the relations, Fig. 8.6 on page 102 illustrates what this constraint describes.

9.5 Avoiding the Redundant Display of Information Walls

With ghost spheres it is theoretically possible to represent the same interface arbitrarily often in Cubes. This is unnecessary and confusing, so we want to rule out certain scenarios as we did for the Cone Tree. On the one hand, the inheritance structure should be displayed not more than once in the same Cube and it should not be displayed in a Cube when no interface is in the Cube's package.

Constraint 9.6. *No two different spheres representing the same interface may reside in the same Cube. If s and t represent the same interface and are not identical, no Cube c can exist that both s and t are members of:*

$$\begin{aligned}
& \forall s, t\ \mathsf{sameInterface}_{s,t} \wedge s \neq t \\
& \quad \rightarrow \neg\exists c\ \mathsf{cubeMemberOf}_{s,c} \wedge\ \mathsf{cubeMemberOf}_{t,c} \\
\Leftrightarrow\ & \mathsf{sameInterface} \cap \overline{\mathsf{I}} \subseteq \overline{\mathsf{cubeMemberOf} \,;\, \mathsf{cubeMemberOf}^{\mathsf{T}}}.
\end{aligned}$$

As opposed to Cone Trees we do not use the extendedCubeMemberOf relation, as spheres are not wrapped inside anything comparable to a Cone Shell.

For the Cone Tree, we argued that the top box of a Cone Tree must be related to at least one solid box, or itself be a solid box, in order to not be a Ghost Tree. We will proceed in the same way for Information Walls. As Walls are not trees, we will not be able to start with one special element (the root) to cover all spheres as it was possible for Cone Trees. We have to check that every sphere that is part of a Wall is connected to at least one solid box or is solid itself.

Constraint 9.7. *Ghost Walls must not exist in a scene. Each sphere that is part of the domain or the range of* connectedByWallPipe *is part of an Information Wall. In this case there must exist a sphere t that is related to s via the reflexive transitive closure (or the other way round) and that is a solid sphere. Again we abbreviate* connectedByWallPipe *with* cbWP*:*

$$
\begin{aligned}
&\forall s\ \mathsf{dom}(\mathsf{cbWP})_s \vee \mathsf{ran}(\mathsf{cbWP})_s \\
&\quad \rightarrow \exists t\ \left((\mathsf{cbWP})^*_{t,s} \vee (\mathsf{cbWP})^*_{s,t}\right) \wedge \mathsf{solidSphere}_t \\
&\Leftrightarrow \forall s\ \left(\mathsf{dom}(\mathsf{cbWP}) \cup \mathsf{ran}(\mathsf{cbWP})\right)_s \\
&\quad \rightarrow \exists t\ \left((\mathsf{cbWP}^*)^{\mathsf{T}} \cup \mathsf{cbWP}^*\right)_{s,t} \wedge \mathsf{solidSphere}_t \qquad (\mathsf{T2,T5}) \\
&\Leftrightarrow \forall s\ \left(\mathsf{dom}(\mathsf{cbWP}) \cup \mathsf{ran}(\mathsf{cbWP})\right)_s \\
&\quad \rightarrow \left[\left((\mathsf{cbWP}^*)^{\mathsf{T}} \cup \mathsf{cbWP}^*\right) ; \mathsf{solidSphere}\right]_s \qquad (\mathsf{T7}) \\
&\Leftrightarrow \mathsf{dom}(\mathsf{cbWP}) \cup \mathsf{ran}(\mathsf{cbWP}) \\
&\quad \subseteq \left((\mathsf{cbWP}^*)^{\mathsf{T}} \cup \mathsf{cbWP}^*\right) ; \mathsf{solidSphere} \qquad (\mathsf{T11}).
\end{aligned}
$$

The single spheres are not comprised in the above constraint. We adopt Cons. 8.9 on page 103 of the the single ghost box discussion (cf. Sect. 8.3) to deal with single ghost spheres:

Constraint 9.8. *No single ghost sphere must exist in a scene. A single sphere is a sphere which is neither in the domain nor in the range of* connectedByWallPipe*. These spheres must be solid. Again,* connectedByWallPipe *is abbreviated with* cbWP*:*

$$
\begin{aligned}
&\forall b\ \overline{\mathsf{dom}(\mathsf{cbWP})}_b \wedge \overline{\mathsf{ran}(\mathsf{cbWP})}_b \rightarrow \mathsf{solidSphere}_s \\
&\Leftrightarrow \overline{\mathsf{dom}(\mathsf{cbWP})} \cap \overline{\mathsf{ran}(\mathsf{cbWP})} \subseteq \mathsf{solidSphere} \qquad (\mathsf{T4,T11}).
\end{aligned}
$$

9.6 Conclusion

As announced before, we will now present a combined conclusion of the Integration of Cone Trees with Cubes *and* Walls with Cubes. Combining

the conclusion makes sense, as the similarities between the two iterations outnumber the differences. *Integration* means for both structures, Cone Trees and Walls, to be brought *inside* the Cube. The challenges we were faced with were both geometrical and semantical.

The geometrical challenge was especially large for Cone Trees. It was not possible to "just stick" the boxes in the Cube; each Cube has a g-Cone that is visualized, and this g-Cone possibly sticks out of the Cube even if all boxes are inside it. We solved this problem by phrasing the Constraints using the Cone Shell that we introduced as an entity in Sect. 5.4. This shell contains the boxes and pipes of a Cone, as well as the matchhing g-Cone. Other notions of complete containment were described but then disregarded, namely, the approach of including a Cone Tree with a hypothetical "Cone Tree Shell" was considered not appropriate as it would consume too much space. The argument that we can rotate a Cone Tree safely inside a Cube only when the complete Cone Tree Shell is included inside a Cube was rebutted by pointing out that "rotation" is not a concept of the language, but a concept of the interaction with Cone Trees inside an editor.

While the Cone Shell (not the Cone *Tree* Shell) solved our containment problem it came with its own issue: when the boxes of a Cone are contained by their Cone Shells they can be contained by a Cube, but they cannot be *members* of the Cube. If we do not know whether or not a box is member of a Cube it is problematic to check any semantical constraint as we showed in Sect. 7.1. To deal with this problem, we introduced extendedCubeMemberOf, a variation of cubeMemberOf which "filtered" the Cone Shells so that a box would be "extended member" of a Cube if it was member of a Cone Shell that was member of that Cube. For spheres such a special treatment was unnecessary since they are not wrapped by any shell. We asserted that any two spheres connected by a Wall pipe must reside inside the same Cube.

The other field of work was to understand what happens if we want to see both package membership and inheritance information together. As we defined that the Cube has to contain the complete Cone Tree or Wall it became possible that some boxes of the Cone Tree or Wall do not belong there semantically. We referred to these boxes and spheres as "ghosts" and we assumed they would be visualized differently than the "solid" (non-ghost) boxes. More interestingly than the visualization of ghost boxes was the implication that one class or interface could be now represented by more than one entity. Before, we were able to interpret all existing boxes as exactly the representation of the different classes. Instead of being a bijection, classOf and interfaceOf were total, surjective mappings. The bijection existed only between classes and solid boxes, respectively interfaces and solid spheres. With the possibility of representing classes and spheres by multiple entities, our goal was to keep the amount of these duplicates to a minimum. As a closer examination of Cone Trees and Walls with ghost entities showed, the necessary amounts of ghosts depends on the amount of different Cubes that

a Cone Tree has to appear in, and that depended on the amount of different packages that the boxes' classes were belonging to. This served as a criterion to decide whether a Cone Tree was supposed to appear in a given Cube or not. The same thoughts applied seamlessly to Walls and ghost spheres.

Although we primarily were concerned with Cone Trees and Walls, we needed to consider the single boxes and spheres to be a complete discussion of the topic. Single boxes and spheres are, by definition, not Cone Trees nor Walls. Most of our findings of the *Cube* chapter (Chapter 7) still applied, and we only needed new constraints to exclude single ghost boxes and single ghost spheres.

Chapter 10

Integration of Cone Trees and Walls

The last "axis of integration" runs between boxes and spheres; respectively, between Cone Trees and Walls. Semantically, this covers the *implements* relationship between classes and interfaces. A class can implement zero to many interfaces and one interface can be implemented by zero to many classes. As before, the current step of integration will include all constraints of earlier chapters, and as the discussion about integration is progressed quite substantially up to this point, not many "degrees of freedom" are left for this aspect.

To show any *implements* relationship we need a pipe. In the course of our thesis this became a routine task, which we will perform in Sect. 10.1. The necessary constraints are developed in Sect. 10.2, accompanied by a detailed discussion about different alternative integration approaches. In Sect. 10.3, the semantical consequences of the integration are analyzed. The experiences we will make with this integration will be summarized in Sect.10.4, where we will also discuss the appropriateness of the Information Cube as a good visualization technique for our purposes.

10.1 The Box-Sphere Pipe

To indicate that a class implements an interface we will (not surprisingly) use a pipe between box and class that we will call the *Box-Sphere Pipe*. As before, we try to avoid the usage of Java related names to keep the application domain separated from the more general possibilities that the SRG has. The next step, as with any pipe, is to introduce a relation for this pipe and constraints that ensure the correct usage.

Relation boxSpherePipe : $\boldsymbol{PIPE \leftrightarrow 1}$. *This vector models the subset of pipes that are intended to run between boxes and spheres:*

$\mathsf{boxSpherePipe}_p$ holds if p is a Box-sphere Pipe.

Relation boxSpherePipeStarts : $\boldsymbol{PIPE \leftrightarrow BOX}$. *This relation contains all pairs of pipes and boxes where the pipe is a Box-sphere Pipe that starts in the box:*

$$\begin{aligned}\mathsf{boxSpherePipeStarts}_{p,b} &:\Leftrightarrow \mathsf{boxSpherePipe}_p \wedge \mathsf{pipeStarts}_{p,b} \\ &\Leftrightarrow (\mathsf{boxSpherePipe} ; \mathbf{L} \cap \mathsf{pipeStarts})_{p,b}.\end{aligned}$$

Relation boxSpherePipeEnds : $\boldsymbol{PIPE \leftrightarrow SPHERE}$. *This relation contains all pairs of pipes and boxes where the pipe is a Box-sphere Pipe that ends in the box:*

$$\begin{aligned}\mathsf{boxSpherePipeEnds}_{p,s} &:\Leftrightarrow \mathsf{boxSpherePipe}_p \wedge \mathsf{pipeEnds}_{p,s} \\ &\Leftrightarrow (\mathsf{boxSpherePipe} ; \mathbf{L} \cap \mathsf{pipeEnds})_{p,s}.\end{aligned}$$

Relation connectedByBoxSpherePipe : $\boldsymbol{SPHERE \leftrightarrow BOX}$. *This relation contains all box-sphere pairs that are connected by a Box-sphere Pipe. A box b is connected to a sphere s (in terms of this relation) if there exists a pipe p that starts in b and ends in s:*

$$\begin{aligned}&\mathsf{connectedByBoxSpherePipe}_{s,b} \\ &:\Leftrightarrow \exists p\ \mathsf{boxSpherePipeEnds}_{p,s} \wedge \mathsf{boxSpherePipeStarts}_{p,b} \\ &\Leftrightarrow \mathsf{boxSpherePipeEnds}^{\mathsf{T}} ; \mathsf{boxSpherePipeStarts}.\end{aligned}$$

Recall that we have the convention to put the "end" entity as the first element in the relation. In the case of connectedByConePipe, this was useful to represent the Cone Tree structure, and from there on we decided to keep the "direction" of the different "connected by" relations consistent.

The obligatory constraint is to ensure that the starts and ends of the pipe are connected to the correct types of entities:

Constraint 10.1. *All Box-sphere Pipes have to start in a box and end in a sphere:*

$$\begin{aligned}&\forall p\ \mathsf{boxSpherePipe}_p \rightarrow \exists b\ \mathsf{pipeStarts}_{p,b} \wedge \mathsf{box}_b \wedge \exists c\ \mathsf{pipeEnds}_{p,c} \wedge \mathsf{sphere}_c \\ &\Leftrightarrow \mathsf{boxSpherePipe} \subseteq \mathsf{pipeStarts} ; \mathsf{box} \cap \mathsf{pipeEnds} ; \mathsf{sphere}.\end{aligned}$$

These tasks were routine and exposed no problems nor any new insights. We turn now towards the specific requirements the integration comes with.

10.2 The Outline of the Integration

Before we go into any details concerning how the integration is performed in terms of constraints, we have to discuss different alternatives of integrations. As it turns out, the fact that entities of different packages need to be integrated will be as challenging as before.

We will go through an iteration of decisions we have to make. We will illustrate each problem, give two alternatives to solve it and justify our decision for one of the choices. Each decision is then formalized by one or more constraints.

10.2.1 Decision 1

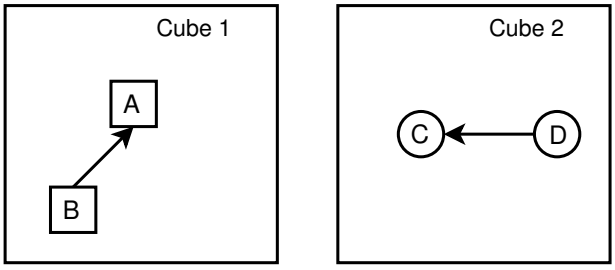
If we want to show an *implements* relation between a box-sphere pair of the same package, there is no problem to display a pipe inside the same Cube. We are also lucky if a box or a sphere is not part of the package, but already resides in the Cube as a ghost box or ghost sphere, because it is part of a Cone Tree or Wall that semantically belongs to the Cube. A new situation occurs if the box and sphere we want to connect are not inside the same Cube. If they are not part of the same Cube, they are not part of the same package and the Cone Tree, respectively Wall, they might be part of has no element in the package of the other.

Fig. 10.1 on the next page illustrates the situation. We want to connect box A with sphere C but they are part of different Cubes and neither the Wall nor the Cone Tree have any duplicates in other Cubes. How can we fix this situation?

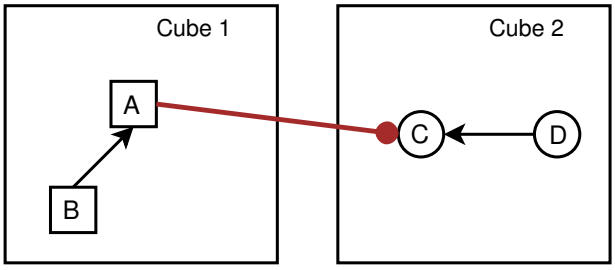
Solution 1.1: we allow the Box-sphere Pipe to run between the Cubes to directly connect the box in the one Cube with the sphere in the other Cube.

Solution 1.2: we do not allow Box-sphere Pipes to run out of Cubes, but drop the strict requirement of no Ghost Trees/Walls and no single ghost boxes/spheres. The Cube with the box shows the sphere and its Wall (if existing) and the Cube with the sphere shows the box and the Cone Tree (if existing).

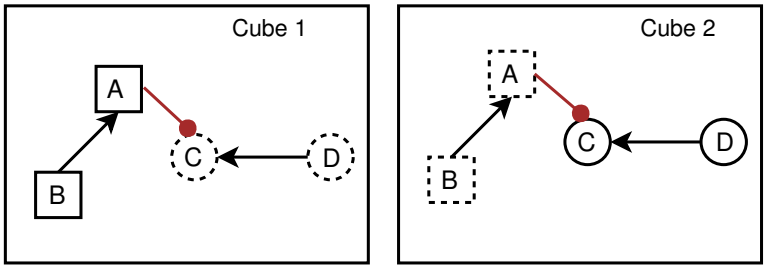
We find that Solution 1.1 is not very attractive. The fact that pipes run between different Cubes might be problematic and confusing if the two Cubes are far away from each other. The user would then have to navigate through the scene to find the matching end of the pipe. If there are several pipes between two Cubes it might also be difficult for the user to associate the different boxes and spheres correctly. Solution 1.2 violates some of our requirements and would also increase the number of boxes/spheres inside a Cube that do not belong there semantically, but it seems less “tangling”



(a) Initial scene, in which an *implements* relation needs to be shown between class A and interface C.



(b) Solution 1.1, connect the entities directly across Cube boundaries.



(c) Solution 1.2, introduce the missing structures in the Cubes mutually for the price of Ghost Trees and Ghost Walls.

Figure 10.1: Two alternatives of showing *implements* relationship between entities of different Cubes.

than the other alternative. We would consequently keep all kind of pipes inside the Cube. Therefore, we chose Solution 1.2.

The results of the earlier discussion regarding the integration pointed in the same direction, but at the latest by this decision the Cube turns from a representation of a Java package into "a container for somewhat related classes and interfaces". It is easily imaginable that an average Cube of an average example project will contain more ghosts than solid entities, that is, more entities that semantically do not belong there than those that do belong there. To formalize Solution 1.2 we expect:

Constraint 10.2. *For each Box-sphere pair it must hold that if they are connected by a Box-sphere Pipe, a Cube exists so that the box at the start and the sphere at the end are member of this Cube:*

$$
\begin{aligned}
&\forall b, s\ \mathsf{connectedByBoxSpherePipe}_{b,s} \\
&\quad \rightarrow \exists c\ \mathsf{extendedCubeMember}_{b,c} \wedge \mathsf{extendedCubeMember}_{s,c} \\
&\Leftrightarrow \forall b, s\ \mathsf{connectedByBoxSpherePipe}_{b,s} \\
&\quad \rightarrow (\mathsf{extendedCubeMember}\ ;\mathsf{extendedCubeMember}^{\mathsf{T}})_{b,s} \qquad (\mathsf{T7}) \\
&\Leftrightarrow \mathsf{connectedByBoxSpherePipe} \\
&\quad \subseteq \mathsf{extendedCubeMember}\ ;\mathsf{extendedCubeMember}^{\mathsf{T}} \qquad (\mathsf{T11}).
\end{aligned}
$$

We also have to drop the strict Constraints 8.8 and 9.7 which disallow Ghost Trees and Ghost Walls, as much as Constraints 8.9 and 9.8, which disallow single ghost boxes and ghost spheres. Instead we need to introduce constraints which verify that Ghost Trees and Walls, single ghost boxes and ghost spheres may exist, but one of their members must be the start or end of a Box-sphere Pipe, showing an *implements* relation.

In order to keep the constraints readable, we will have two constraints; one for Ghost Trees, and one for Ghost Walls. For the constraints we can reuse part of our work we did for the "no Ghost Trees" constraint (Cons. 8.8), namely the identification of Ghost Trees and Walls. We can reach all boxes of a Cone Tree by the reflexive-transitive closure

$$(\mathsf{connectedByConePipe} \cup \mathsf{connectedByConePipe}^{\mathsf{T}})^*.$$

Due to Constraint 10.2, we know that the two ends of the Box-sphere Pipe must reside in the same Cube so we do not need to demand it in the constraints.

Constraint 10.3. *Ghost Trees are allowed as long as at least one of the boxes connects to a sphere via a Box-sphere Pipe. For reasons of compactness*

we will substitute connectedByConePipe *with* cbCP *as we did before:*

$$\begin{aligned}
&\forall b\ \neg\exists c\ (\mathsf{cbCP} \cup \mathsf{cbCP}^{\mathsf{T}})^*_{b,c} \wedge \mathsf{solidBox}_c \\
&\quad \rightarrow \exists d\ (\mathsf{cbCP} \cup \mathsf{cbCP}^{\mathsf{T}})^*_{b,d} \wedge \mathsf{ran}(\mathsf{connectedByBoxSpherePipe})_d \\
&\Leftrightarrow \forall b\ \overline{\left((\mathsf{cbCP} \cup \mathsf{cbCP}^{\mathsf{T}})^* \,;\mathsf{solidBox}\right)}_b \\
&\quad \rightarrow ((\mathsf{cbCP} \cup \mathsf{cbCP}^{\mathsf{T}})^* \,;\mathsf{ran}(\mathsf{connectedByBoxSpherePipe}))_b \qquad (\mathsf{T7},\mathsf{T6}) \\
&\Leftrightarrow \overline{(\mathsf{cbCP} \cup \mathsf{cbCP}^{\mathsf{T}})^* \,;\mathsf{solidBox}} \\
&\quad \subseteq (\mathsf{cbCP} \cup \mathsf{cbCP}^{\mathsf{T}})^* \,;\mathsf{ran}(\mathsf{connectedByBoxSpherePipe}) \qquad (\mathsf{T11}).
\end{aligned}$$

Now we state the same for Ghost Walls:

Constraint 10.4. *Ghost Walls are allowed as long as at least one of the spheres connects to a box via a Box-sphere Pipe. For reasons of compactness we will substitute* connectedByWallPipe *with* cbWP *as we did before:*

$$\begin{aligned}
&\forall s\ \neg\exists t\ (\mathsf{cbWP} \cup \mathsf{cbWP}^{\mathsf{T}})^*_{s,t} \wedge \mathsf{solidSphere}_t \\
&\quad \rightarrow \exists u\ (\mathsf{cbWP} \cup \mathsf{cbWP}^{\mathsf{T}})^*_{s,u} \wedge \mathsf{dom}(\mathsf{connectedByBoxSpherePipe})_u \\
&\Leftrightarrow \overline{(\mathsf{cbWP} \cup \mathsf{cbWP}^{\mathsf{T}})^* \,;\mathsf{solidSphere}} \\
&\quad \subseteq (\mathsf{cbWP} \cup \mathsf{cbWP}^{\mathsf{T}})^* \,;\mathsf{dom}(\mathsf{connectedByBoxSpherePipe}).
\end{aligned}$$

The same kind of requirement applies for single ghost boxes and ghost spheres:

Constraint 10.5. *Single ghost boxes are allowed, but only if they have a sphere connected to them via a Box-sphere Pipe. Box b is a* single *ghost box if it is a ghost box and neither in the range nor in the domain of* connectedByConePipe*:*

$$\begin{aligned}
&\forall b\ \mathsf{ghostBox}_b \wedge \overline{\mathsf{dom}(\mathsf{cbCP})}_b \wedge \overline{\mathsf{ran}(\mathsf{cbCP})}_b \\
&\quad \rightarrow \mathsf{ran}(\mathsf{connectedByBoxSpherePipe})_b \\
&\Leftrightarrow \forall b\ \left(\mathsf{ghostBox} \cap \overline{\mathsf{dom}(\mathsf{cbCP})} \cap \overline{\mathsf{ran}(\mathsf{cbCP})}\ \right)_b \\
&\quad \rightarrow \mathsf{ran}(\mathsf{connectedByBoxSpherePipe})_b \qquad (\mathsf{T4}) \\
&\Leftrightarrow \mathsf{ghostBox} \cap \overline{\mathsf{dom}(\mathsf{cbCP})} \cap \overline{\mathsf{ran}(\mathsf{cbCP})} \\
&\quad \subseteq \mathsf{ran}(\mathsf{connectedByBoxSpherePipe}) \qquad (\mathsf{T11}).
\end{aligned}$$

The same applies for single ghost spheres:

Constraint 10.6. *Single ghost spheres are allowed, but only if they have a box connected to them via a Box-sphere Pipe:*

$$\begin{aligned}&\forall b\ \mathsf{ghostSphere}_b \wedge \overline{\mathsf{dom}(\mathsf{cbWP})}_b \wedge \overline{\mathsf{ran}(\mathsf{cbWP})}_b \\ &\quad \rightarrow \mathsf{dom}(\mathsf{connectedByBoxSpherePipe})_b \\ &\Leftrightarrow \mathsf{ghostSphere} \cap \overline{\mathsf{dom}(\mathsf{cbWP})} \cap \overline{\mathsf{ran}(\mathsf{cbWP})} \\ &\quad \subseteq \mathsf{dom}(\mathsf{connectedByBoxSpherePipe}).\end{aligned}$$

10.2.2 Decision 2

The next question arising is: do we allow Ghost Walls or Ghost Trees to be inside a Cube under any circumstances? To understand this question, look at Fig. 10.2 on the following page, where we see two approaches differing in how they treat Ghost Boxes (the same would apply for Ghost Walls).

Again, two solutions are offered:

Solution 2.1: yes, for any box and sphere (solid and ghost): allow the missing Ghost Tree or Ghost Wall inside the Cube. The same would apply for single ghost boxes or spheres.

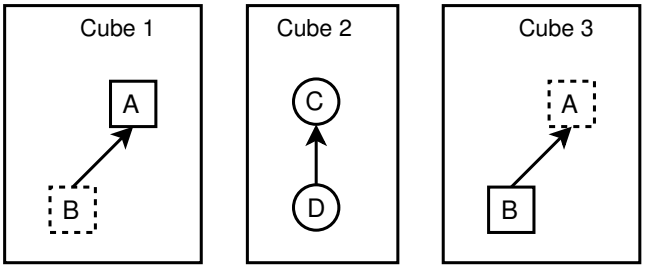
Solution 2.2: only allow a missing Ghost Tree, Ghost Walls or single ghost boxes or spheres inside a Cube if the box or sphere is part of an "implements" relation.

As before, we decide for the second solution, because Solution 2.1 would come at the price of a huge "blow-up" of Cubes, as we would recursively introduce ghosts inside the Cubes. We originally introduced the ghosts as a "necessary evil" to show inheritance and package membership at the same time. We find it inappropriate to burden this additional responsibility upon an auxiliary construct.

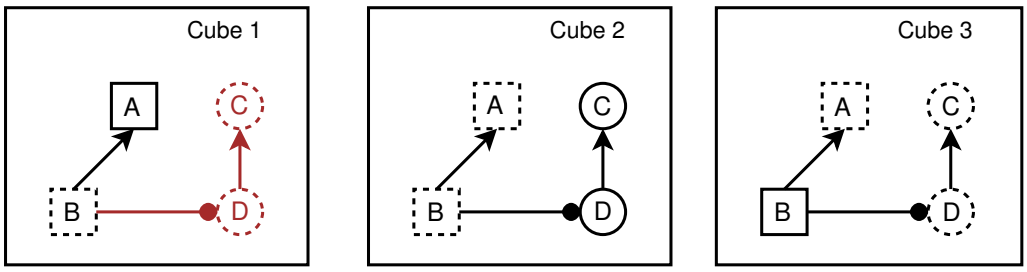
Choosing Solution 2.2, we have a predictable amount of new ghosts added to each Cube: either *zero*, because the box and the sphere are already in the same Cube, or *two*, once a Cone Tree in the sphere's Cube and once a Wall in the box' Cube. The second answer reduces the amount of Ghost Trees to a necessary minimum. As before, our decision does not automatically result in a constraint because it is smart to make another decision first which will allow us to find a constraint that covers both decisions.

10.2.3 Decision 3

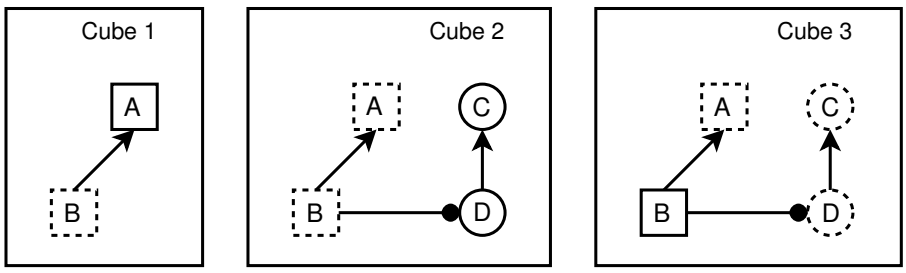
A remaining question is: even if we do not introduce new Ghost Trees or Walls for ghost spheres and boxes, do we show the implements relation between two ghosts if they are part of the same Cube already? Such a situation might occur if the Cone Tree and the Wall are in the same Cube



(a) Initial scene, in which an *implements* relation needs to be shown between class B and interface D.



(b) Solution one, Show the *implements* relation for all occurrences of B and C. This particularly includes occurrences of B as a ghost box in Cube 1.



(c) Solution two, only show the *implements* relationship for the solid occurrences of the class or the interface.

Figure 10.2: Two alternative policies for allowing Ghost Trees (and Ghost Walls) in a Cube.

because some of the boxes and spheres belong to the Cube and there exists an implements relation between two entities that are not part of the Cube. Fig. 10.3 on the next page shows such a situation.

Two plausible answers to the question are:

Solution 3.1 yes, for any box and sphere (solid and ghost): show all *implements* relationships between any boxes and spheres that are already part of the Cube. That includes box-sphere pairs where both entities are ghosts.

Solution 3.2: no, show the *implements* relationship only for box-sphere pairs where at least one of the entities is solid, even when two ghost entities are already inside the Cube.

In the context of our earlier decisions, we find Solution 3.2 more consequent. If we do not show Box-sphere Pipes for some ghost-ghost pairs, we should not show them for any pairs of ghost entities. That way $\mathrm{J^3D_L}$ enforces a clear policy: if the user wants to know about all relations of a class or an interface, the user must navigate to the solid box or sphere that represents it. A ghost box is merely a "proxy", and the user knows that the original representative can be found elswhere. Together with Solution 2.2, where we said that a ghost box or sphere cannot be the reason to introduce a Ghost Wall or Tree into an Cube we get:

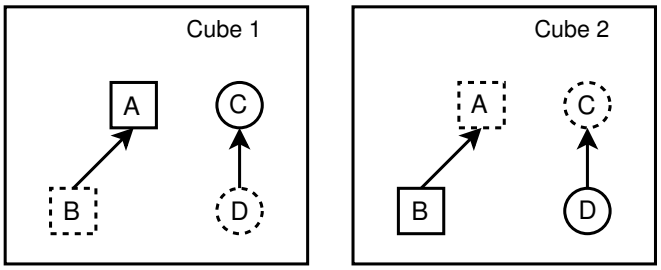
Constraint 10.7. *Box-sphere pipes must either have a solid entity at their start or at their end. For any box and sphere connected by a Box-sphere Pipe, it must hold that not both are ghosts:*

$$\begin{aligned}
&\forall b, s\ \mathsf{connectedByBoxSpherePipe}_{s,b} \rightarrow \neg(\mathsf{ghostBox}_b \wedge \mathsf{ghostSphere}_s) \\
&\Leftrightarrow \forall b, s\ \mathsf{connectedByBoxSpherePipe}_{s,b} \rightarrow (\overline{\mathsf{ghostSphere}\ ;\mathsf{ghostBox}^{\mathsf{T}}})_{s,b} \quad (\mathsf{T2},\mathsf{T7}) \\
&\Leftrightarrow \mathsf{connectedByBoxSpherePipe} \subseteq \overline{\mathsf{ghostSphere}\ ;\mathsf{ghostBox}^{\mathsf{T}}} \quad (\mathsf{T11}).
\end{aligned}$$

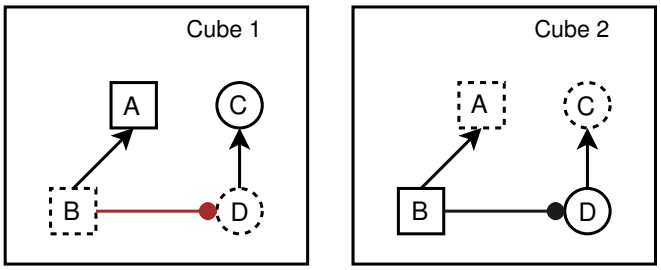
10.3 Semantical Considerations

With the above decisions made, we have to still describe *how* the implements relationship is connected to the Box-sphere Pipes in the scene. So far we have fixed only the consequences of dropping the "no Ghost Tree / no Ghost Wall" constraints and our decision to never show the Box-sphere Pipes between two ghost entities.

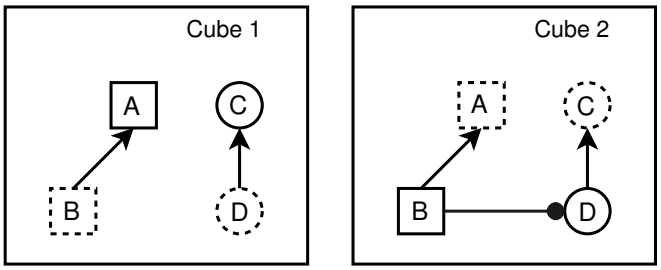
Constraint 10.8. *If class c implements interface i there must exist a box b and a sphere s connected by a Box-sphere Pipe. The box must (directly or indirectly) represent class c and the sphere must (directly or indirectly)*



(a) Initial scene, in which an *implements* relation needs to be shown between class B and interface D.



(b) Solution 1, show the *implements* relation also between two ghosts if the box and sphere are part of the Cube already.



(c) Solution 2, show the *implements* relationship for the solid occurrences of the class or the interface. Do not show the relationship between two ghosts, even if two ghost entities are already existing in the Cube.

Figure 10.3: Two alternatives policies for showing connections between ghost boxes and spheres.

represent interface i. The transformation is similar to semantical constraints at other places of the thesis:

$$\begin{aligned}
&\forall c, i\ \mathsf{implements}_{c,i} \\
&\quad \leftrightarrow \exists b, s\ \mathsf{extendedClassOf}(b) = c \wedge \mathsf{extendedInterfaceOf}(s) = i \\
&\quad \wedge\ \mathsf{connectedByBoxSpherePipe}_{s,b} \\
&\Leftrightarrow \mathsf{implements} \\
&\quad = \mathsf{extendedClassOf}^{\mathsf{T}} \,;\mathsf{connectedByBoxSpherePipe}^{\mathsf{T}} \,;\mathsf{extendedInterfaceOf}.
\end{aligned}$$

Together with the constraints from above, we know that such a pipe connects only box and sphere inside the same Cube and never connects two ghost entities. With all our experience in the integration made so far we are not so naive to assume that this constraint alone is sufficient. Since there is no one-to-one relationship between boxes and classes or interfaces and spheres, we have to ensure that the Box-sphere Pipes appear everywhere where expected.

This situation might remind one of the Cone Pipes (and Wall Pipes) for which we had to ensure that they appeared on *all* corresponding Cone Trees. The difference between the current problem and the one with the Cone Trees is: we excluded some situations where Box-sphere Pipes could have been shown, namely those where both start and end of the pipe are ghosts.

While this may sound like a "more complicated" constraint, a little reflection reveals that it will be actually easier than it was for the Cone Trees. Only two situations can occur that are in accordance with our earlier determinations:

Case 1: class and interface are in the same package. Exactly one pipe can be validly shown. It runs between the solid box and the solid sphere in the Cube that represents the package of class and interface. Any other occurrence of a box and a sphere representing this class and this interface will be one of two ghost entities.

Case 2: class and interface are in different packages. Exactly two pipes can be shown validly. One in the Cube that represents the package of the class (where the appropriate box is solid) and one in the Cube that represents the package of the interface (where the sphere is solid). In any other Cube where a box and a sphere appear that represent the class and the interface the two entities will be ghosts.

In the first case, Cons. 10.8 is sufficient, as it is enforcing the existence of the only possible pipe. For the second case, we need another requirement; but it is easy to state since we only have two possible corresponding pairs,

one a solid box and ghost sphere and the other pair a ghost box and solid sphere. With Cons. 10.8 we know that at least one of them must exist and the constraint must ensure that the other one is also existing:

Constraint 10.9. *If b is a ghost box and s is a solid sphere or the other way round and both are connected by a Box-sphere Pipe there must exist another box c and another sphere t so that b is ghost box of t or reversed as well as t must be ghost box of s or reversed and c and t must be connected by a Box-sphere Pipe. For the sake of readability we will use the following abbreviations:* cbBSP *for* connectedByBoxSpherePipe, gBO *for* ghostBoxOf *and* gSO *for* ghostSphereOf*:*

$$
\begin{aligned}
&\forall b, s\ (\mathsf{ghostBox}_b \wedge \mathsf{solidSphere}_s \vee \mathsf{solidBox}_b \wedge \mathsf{ghostSphere}_s) \\
&\qquad \wedge \mathsf{cbBSP}_{s,b} \\
&\quad \rightarrow \exists c, t\ (\mathsf{gBO}_{b,c} \vee \mathsf{gBO}_{c,b}) \wedge (\mathsf{gSO}_{t,s} \vee \mathsf{gSO}_{s,t}) \wedge \mathsf{cbBSP}_{t,c} \\
\Leftrightarrow\ &\forall b, s\ (\mathsf{ghostBox} \,;\mathsf{solidSphere}^{\mathsf{T}} \cup \mathsf{solidBox} \,;\mathsf{ghostSphere}^{\mathsf{T}})_{b,s} \\
&\qquad \wedge \mathsf{cbBSP}^{\mathsf{T}}_{b,s} \\
&\quad \rightarrow \exists c, t\ (\mathsf{gBO} \cup \mathsf{gBO}^{\mathsf{T}})_{b,c} \wedge (\mathsf{gSO} \cup \mathsf{gSO}^{\mathsf{T}})_{t,s} \wedge \mathsf{cbBSP}^{\mathsf{T}}_{c,t} \qquad (\mathsf{T7,T5,T2}) \\
\Leftrightarrow\ &\forall b, s\ \big((\mathsf{ghostBox} \,;\mathsf{solidSphere}^{\mathsf{T}} \cup \mathsf{solidBox} \,;\mathsf{ghostSphere}^{\mathsf{T}}) \\
&\qquad \cap \mathsf{cbBSP}^{\mathsf{T}}\big)_{b,s} \\
&\quad \rightarrow \exists c, t\ (\mathsf{gBO} \cup \mathsf{gBO}^{\mathsf{T}})_{b,c} \wedge (\mathsf{gSO} \cup \mathsf{gSO}^{\mathsf{T}})_{t,s} \wedge \mathsf{cbBSP}^{\mathsf{T}}_{c,t} \qquad (\mathsf{T4}) \\
\Leftrightarrow\ &(\mathsf{ghostBox} \,;\mathsf{solidSphere}^{\mathsf{T}} \cup \mathsf{solidBox} \,;\mathsf{ghostSphere}^{\mathsf{T}}) \\
&\qquad \cap \mathsf{cbBSP}^{\mathsf{T}} \\
&\quad \subseteq (\mathsf{gBO} \cup \mathsf{gBO}^{\mathsf{T}}) \,;\mathsf{cbBSP}^{\mathsf{T}} \,;(\mathsf{gSO} \cup \mathsf{gSO}^{\mathsf{T}}) \qquad (\mathsf{T11}).
\end{aligned}
$$

One might wonder if the constraint works correctly; i.e., if the right combination of solid and ghosts are enforced by the constraint. It is easy to see that this is the case.

Theorem 10.1. *If the the left side of Cons 10.9 and the implication as a whole are true for box b and sphere s it must hold that:*

- *if b is a ghost box then s is a solid sphere and c must be the solid box of c and t is the ghost sphere for s,*
- *if b is a solid box then s is a ghost sphere and c is a ghost box for c and t is the solid box for s.*

Proof. We show the proof for b being the ghost box and s the solid sphere, the other combination is completely analogous.

$$
\begin{aligned}
&\mathsf{ghostBox}_b \wedge \mathsf{solidSphere}_s && \text{(Assumption)}\\
&\Rightarrow \neg\mathsf{solidBox}_b \wedge \neg\mathsf{ghostSphere}_s &&\\
&\Rightarrow \neg\mathsf{ghostBoxOf}_{c,b} \wedge \neg\mathsf{ghostSphereOf}_{s,t} &&\\
&\Rightarrow \mathsf{ghostBoxOf}_{b,c} \wedge \mathsf{ghostSphereOf}_{t,s} && \text{(Cons. 10.9).}
\end{aligned}
$$

□

Hence the constraint handles both possibilities mutually exclusive and therefore functions as expected. The above constraint is also valid for *single* ghost boxes and ghost spheres as it is not referring to Cone Trees or Walls.

10.4 Conclusion

What were our experiences with the last part of integrating boxes, spheres, and Cubes? The *implements* relation is pipe indicated and we were able to deal with the tasks related to it routinely. It seems as if our set of relations and constraints for pipes is relatively stable throughout the different sections. We also were able to deal with the problems that ghost entities and the correspondence of Ghost Trees and ghost Walls. A lot of the insights from earlier sections could be applied successfully.

The concluding judgement on the integration in its completeness brings up one negative experience: while we were able to handle the relation-algebraic description of ghost boxes and spheres with ease, they seem to be inappropriate for this degree of integration from a visual and cognitive point of view. The Cube morphed from a representation of packages into a mere container of otherwise related boxes and spheres. Some of the boxes and spheres might even belong into it originally but it is easy to envision common modeling scenarios where more ghost than solid entities appear in a Cube. The Cube lost its significance in terms of showing package membership.

Next to that, the scene is also subject to a huge "blow-up" where much more Cone Trees and Walls are inside a Cube than intended. This is due to the fact that theoretically an inheritance hierarchy with n classes can lead to n Cone Trees in different Cubes, plus numerous Ghost Trees when implements relationships are to be shown. For the same reason, a package with m classes and interfaces can hold more than m Cone Trees and Walls. The approach of integrating inheritance package membership and implements relations leads to a huge redundancy and to huge dimensions of Cubes. Not only has a Cube to host all its Cone Trees and Walls it also has to be big enough to include its sub-Cubes, which in return have Cone Trees, Walls and other Cubes themselves.

Though it is possible to describe relation-algebraically as shown in the previous sections, we doubt that the integration done as described would have any benefit for the purpose of modeling and understanding Java structures. It is hard to give alternatives at this point of which we can be confident that they do not carry similar problems. The ghost entities appeared as a by all means valid solution to solve the integration problem and only a in-depth analysis uncovered the mentioned problems. Any suggestion we can make at this point might fall short of combining the package and inheritance view.

From a pessimistic point of view it might be best to not show inheritance and package view at the same time. Our point of view is a little more optimistic as we think that the choice of visualization is the problem, especially the "sourrounding" metaphor of the Cubes. It might be worth trying to visualize the Cubes as entities but to show the membership between members and the Cube by pipes. That way no duplicates of boxes and spheres need to be introduced. Another alternative could include to color the boxes of a Cone Tree and the spheres of a Wall according to their package membership. Such an approach would admit though that three dimensions are not enough to combine the different views and a fourth dimension needs to be incorporated.

Part IV

Beyond the Language Definition

Chapter 11

Consequences of the Language Design for Implementation

With the integration of Cubes, Cone Trees and Walls we have concluded the definition of J^3D_L. In this chapter we want to take a look at the implementation. Our goal is not to give concrete advice on how to program an 3D software editor using specific technologies. This has been discussed in [DFSH04], [Tch04], or [Roh04]. We want to remain on a conceptual level and merely point out the general consequences that the design of J^3D_L comes with.

First we introduce different usage scenarios in Sect. 11.1 and argue why one of them is closest to J^3D_L. This scenario will be described in more detail in Sect. 11.2. In Sect.11.3 we examine situations in which it might possible to reduce the amount of constraints that need to be checked.

11.1 Usage Scenarios

In the last chapters we have seen, how J^3D_L connects the SRG and the ASG with each other. A subset of relations and constraints were introduced, that dealt exclusively with the connection. Form an abstract point of view different usage scenarios are imaginable, which differ in how they interpret the connection of the ASG and the SRG. We want to introduce them here to discuss if they are covered by the current design of J^3D_L and also what would need to be changed in order to cover them. Three substantially different scenarios exit:

Scenario 1: From a 3D scene to Java code. The fundamental relations of the SRG are provided and if the constraints are valid, the ASG relations can be computed.

Scenario 2: From Java code to (parts of) a 3D scene. With the relations on the ASG level we check the validity of the Java code and parts of the SRG are computed.

Scenario 3: Matching Java code and 3D scenes. Both ASG and SRG are provided independently and are checked if they match; which means, if the Java code represented by the ASG relations is reflected in the 3D scene represented by the SRG.

In each scenario the constraints regarding the relation between ASG and SRG is interpreted differently: in Scenarios 1 and 2 the right-hand side (ASG or SRG) of the constraints is used to compute the relations on the left-hand side and in scenario three the constraints are used to compare the two sides. Following is a more detailed discussion of the three scenarios.

11.1.1 Scenario 1: From the Scene to the Code

Looking at the three scenarios, the first scenario is the one that is covered the best by the current design of $\mathrm{J^3D_L}$. This comes as no surprise, as it is the scenario we motivated the effort of creating the language for in the first place. The goal is to declaratively specify the conditions under which a 3D scene is valid (both visually and semantically) to obtain valid Java code (skeletons). In the thesis the constraints are designed to serve this purpose, so that the ASG relations are declared to equal certain relations of the SRG. Constraint 8.5 on page 99, for example, states:

$$\mathsf{inheritsClass} = \mathsf{extendedClassOf}^{\mathsf{T}} ; \mathsf{connectedByConePipe}^{\mathsf{T}} ; \mathsf{extendedClassOf}.$$

which means that one class inheriting from another class is equal to two boxes being connected by a Cone Pipe. How can we compute the inheritance relation out of the right hand expression? In this scenario we expect the SRG relations as given, in this case connectedByConePipe. The connection between ASG and SRG will have to be computed relation-algebraically as well. This can be done by defining

$$\mathsf{class} := \mathsf{solidBox}, \quad \text{and} \quad \mathsf{classOf} := \mathsf{I}.$$

As classOf is required to be bijective and runs between classes and solid boxes, we can easily construct it by defining class to be a copy of box and classOf to be the identity between the two vectors. As we have no relation on the ASG side we are free in associating classes with boxes and the above solution seems to be the easiest way to do so. As ghostBoxOf is already a fundamental relation in this scenario we have all parts together to compute extendedClassOf. The computation of inheritsClass is no problem afterwards.

This process can be applied for other semantical constraints as well, allowing us to compute the complete ASG out of the SRG.

11.1.2 Scenario 2: From the Code to the Scene

In Scenario 2 the situation is reversed. We will use the same example as in Scenario 1. This time we are equipped with a given inheritsClass relation as part of the ASG. After a slight modification of the above example we could state:

$$\mathsf{connectedByConePipe} = \mathsf{extendedClassOf}^{\mathsf{T}} ; \mathsf{inheritsClass}^{\mathsf{T}} ; \mathsf{extendedClassOf}.$$

Now we have to compute connectedByConePipe out of the right side of the constraint, and again the connection between ASG and SRG (extendedClassOf) has to be established. Instead of stating

$$\mathsf{class} := \mathsf{solidBox},$$

we state the reverse:

$$\mathsf{solidBox} := \mathsf{class}.$$

As the SRG is not defined in this scenario we can arbitrarily create (solid) boxes and their associations to classes as long as we obey the relevant constraints.

Unfortunately this still does not tell us which ghost boxes we need, nor how they are related to the solid ones. In the previous chapters we assumed (thinking in terms of Scenario 1) that the information about ghosts is provided by the editor as part of the SRG. To cope with the ghost entities new relations would need to be developed, which construct the ghost box information out of the inheritsClass and the packageMemberOf relations.

Even with these new relations we would not obtain a complete SRG. While entities such as boxes, spheres, pipes and Cubes can be computed as well as the pipes between them, the spatial relations remain incomplete. For example, we know (Cons. 5.9) that

$$\mathsf{connectedByConePipe} \subseteq \mathsf{above}.$$

However, to get a complete 3D scene, we would need a separate layout component that puts actual numerical values on top of the spatial relations, and that decides the relative positions of those parts of the scene that are not constraint. As an example, consider the relative alignment of two Cone Trees that are not related with each other. No constraints exists that gives us information about their relative spatial arrangement.

It is hard to estimate if such a layout algorithm would benefit from the partial information such as which boxes are above which, or which spheres are on the same height as other spheres. It seems more practical to only compute the structural relations of the SRG by relational algebra, and leave the construction of the Physical Layout and the remaining SRG relations to the algorithm.

Also, the validity of the Java code represented by the ASG relation can only be checked indirectly through the constraints on the SRG level. For this scenario it might be more desirable to check the validity of the Java relations directly on the ASG level before translating it into a 3d scene. In this case new constraints on the ASG level would need to be developed.

In summary we can see that Scenario 2 is not as well covered by the definition of J^3D_L as Scenario 1.

11.1.3 Scenario 3: Matching Scene and Code

In the third scenario we could check if the ASG is consistent with the structural relations on the SRG. In this case we actually interpret the equal sign between the left side and the right side of a semantical constraint as something we have to verify. The challenge of this scenario is to find a mapping between Java elements and SRG entities so that the constraints are true, which reminds one of the constraint solving programs that are part of the Artificial Intelligence research. It seems to be a very specialized scenario in which one has a 3D scene together with some Java code and wants to check whether the given scene represents the given Java code.

11.2 The Scene-to-Code Usage Scenario

In this section we want to detail Scenario 1 a little more. In the previous section we have looked into what must change for the individual constraints, and in this section we want to focus on the larger picture, which means to look into how different parts of the application interact which each other. We identified four parts of such an application:

The 3D editing environment to create and modify the scene. This editor will most likely use some sort of framework (e.g. Java3D [Ja3]) to represent the scene in.

A scene parser that translates the framework's objects describing the scene into SRG relations.

The Relational engine that computes the internally used relations and checks the constraints.

A relation parser which can translate the ASG relations into Java source code or some other representation for Java code.

We again want to point out that these components do not necessarily have to correspond with components in the implementations, they are merely logical groupings of tasks. The interaction of these components is detailed in Fig. 11.1 on the next page.

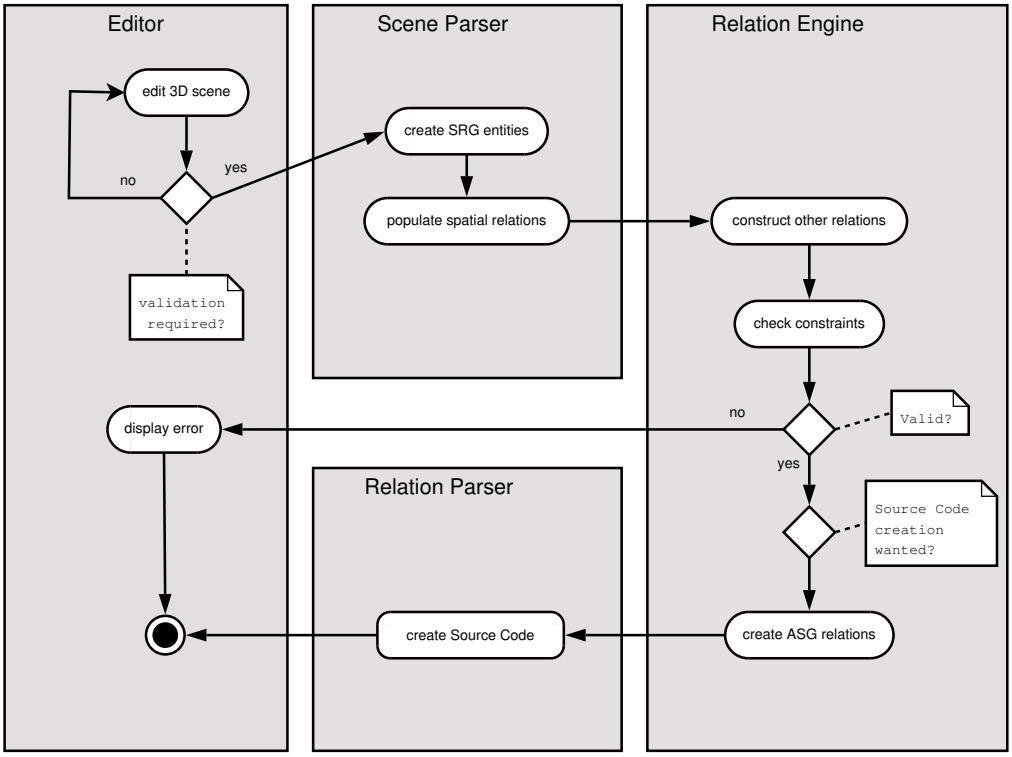


Figure 11.1: Activity Diagram showing the checking of constraints and the creation of source code

11.2.1 The Editor

As we can see, the diagram suggests that the editor does not check every change, but triggers the checking of constraints under certain conditions. With this interaction policy the editor can allow scenes that are (temporarily) invalid. When the user, for example, drags a box from one Cube to another it will have to leave the old Cube and enter the new one. During leaving and entering the box will overlap with the Cube but it is not contained by it, which is a violation of Constraint 7.1 on page 81. If the editor would trigger a complete constraint check after every user event ("mouse moved"), it would render it impossible to move a box outside of a Cube by dragging as it would run into a constraint violation as soon as the box reaches the Cube's edge. To avoid this, the editor could let only certain events trigger a constraint check, such as saving or loading a scene or when the user explicitly wishes to. Needless to say that a checking of all constraints after every user action might be costly and slow down the editor. We will discuss options to reduce the frequency of checking constraints and to reduce the number of constraint in Sect. 11.3.

11.2.2 The Scene Parser

When the editor decides to check the scene, the scene parser comes into play. Its purpose is to translate the 3D scene into SRG relations. It is aware of the specific framework the editor has been implemented in and the relational engine that is used ([Tch04]). These relations are the vectors describing the set of entities (box, sphere, and so on) and fundamental relations between them (pipeStarts, above, contains).

The scene parser must also resolve the geometrical problems described in Chapter 4, such as the measuring of distances.

11.2.3 The Relational Engine

The relational engine is the "heart" of the language checker. Before showing how it interacts with other components we want to give a short introduction in one relational engine that is used in the Vise3D context: RelView[RHP05, BS97].

RELVIEW

The homepage [RHP05] describes RelView as

> *. . . an interactive tool for computer-supported manipulation of relations represented as Boolean matrices or directed graphs, especially for prototyping relational specifications and programs.*

In RelView, relation-algebraic expressions can be stated as strings composed out of the names of the relations and the operations in infix notation. For the two relations, R and S, the relation $\mathsf{R};\overline{\mathsf{S}}$ can be computed by `R * -S` with `*` representing the composition operation and `-` the negation of S.

The constraints and relations that we develop in the course of this thesis can be fed to RelView in the form of tests. Tests are comparisons between one or two relations regarding *inclusion* and *equality*. For single relations the tests *empty*, *univalent*, *total*, *injective*, and *surjective* are predefined.

If, for example, one of our constraints states that no pair that is part of the leftOf relation can be part of the rightOf relation at the same time, we get a constraint:

$$\mathsf{leftOf} \subseteq \overline{\mathsf{rightOf}},$$

which we can state using the predefined inclusion function `incl()`:

```
inlc(leftOf, -rightOf).
```

The outcome of the comparison is either *true* or *false*, which is represented by the universal relation $\mathbf{L} : 1 \leftrightarrow 1$ or $\mathbf{O} : 1 \leftrightarrow 1$.

To use RelViewas part of an application, without its GUI, the library Kure [KHP05] exists, which provides the relational functions of RelView as a C library. On top of Kure a wrapper was developed [Szy03], that makes the functionality of Kure available for Java programs.

The Role of the Relational Engine

Equipped with the fundamental relations, the actual relation engine can start to work. The first step is to create relations that are derived from the fundamental ones, such as connectedByConePipe or extendedCubeMembership, which are necessary to check the constraints. Then the actual checking of constraints can happen. If a constraint is not met, this is communicated to the editor, which in return can inform the user of the problem.

If all constraints are valid, the ASG relations can be computed to create Java code. This is only done, if the user decides for it. If the user simply wants to know whether or not the scene is valid as an intermediate feedback it is not necessary to create Java code for it. In the case that creation of Java code is wanted, the ASG relations are derived from the SRG ones. We have said before in Sect. 11.1 that in this scenario the semantical constraints are not considered constraints but are used as equations to compute the ASG relations, such as implements.

11.2.4 The Relational Parser

The last step, done by the relational parser, is to turn the ASG relations into Java code. While we do not want to give a detailed description of this process, it seems to be straightforward:

The ASG elements correspond to files (classes and interfaces) and folders (packages). They are related to each other by strings in the source code ("extends", "implements"), that are added to the files. For each element in class and interface a file is created (the filename will be part of the scene but is not part of the model) and for each of the different relations between classes and interfaces the appropriate text is added to the file. For example if $\mathsf{inheritsClass}_{b,c}$ is true, the Java class file for b gets an entry ... `extends C`. The folders for the packages are created out of the packageMemberOf relation, which includes the package hierarchy. The class or interface c gets a `package p.q` string added if $\mathsf{packageMemberOf}_{q,p}$ and $\mathsf{packageMemberOf}_{c,q}$ hold. Reversely, the relations could be created out of the source code by analyzing the code for the appropriate strings.

If the editor is implemented as part of Eclipse, as in [DFSH04] or [Tch04] the ASG gets translated into an intermediate representation instead of the class files.

11.3 Considerations Regarding the Implementation of Constraints

So far we have seen that the constraints can be subject to change depending on the scenario, and we have also seen that in the case of Scenario 1 (from the scene to the code) the semantical constraints are not used to check the scene but are used to compute the ASG out of the SRG.

In the activity diagram of Fig. 11.1 on page 133, there was only one activity called "check constraints". We want to discuss how and which constraints need to be checked based on different assumptions:

- checking of constraints depending on the features of the editor,
- restrict certain groups of constraints depending on the desired outcome,
- the problems to reduce the checking of constraints in dependency from certain activities of the user.

Functions of the Editor

When we designed $\mathrm{J^3D_L}$ we aimed to be editor independent and hence made no assumptions about the functionality the different editors might provide. When talking about implementation-related concerns we have to take the editors functionality into consideration. In Chapter 4, we specified the the requirements for the fundamental relations. The editor is supposed to provide these relations and we will not check the requirements as we assume that the editor (or more specifically the scene parser, cf. Sect. 11.2) is implemented correctly and the relations possess the expected properties.

This thought extends to other functionalities. If an editor is implemented in a way that makes it impossible for atomic entities (boxes, spheres) to overlap or does not allow that pipes dangle (do not end and start in entities), there is no need for the relation engine to check these constraints. Checking constraints that match features of an editor means to check the implementation of the editor and not properties of a particular 3D scene. It is therefore an important task when implementing the constraints to carefully examine the functionalities that an existing editor has or a yet-to-be-programmed editor will have and to exclude the constraints that are covered by features of an editor.

This thought can also be understood in a different way: if someone wants to program an editor that is specifically intended to support J^3D_L-modeling, the constraints (at least the part describing the constaint textually) are requirements for the functionalities of the editor. In doing so, the amount of mistakes that a user can make in terms of the language's syntax are reduced, which is a true benefit for using such an editor.

Output Dependent Checking

Another way to refine the checking of constraints is to group them by what part of the language they are checking and to check only some of these groups if the user wishes. During the development of constraints we had a rough separation into structural, geometrical and semantical constraints. We know now that the semantical constraints are not checked in the chosen scenario but are used to compute the ASG. This leaves us with the structural, and the geometrical, constraints.

If the goal is to just create valid Java code, most of the geometrical constraints do not need to be checked. It is, for example, not necessary for a parent box to be above its child boxes just to represent a valid inheritance structure of classes. On the other hand, the geometrical constraints (membership) regarding Cubes are highly important for the underlying Java code, as they represent the package membership. Also, the constraints regarding ghost entities are necessary to ensure valid Java code. An implementation could reflect this by having different subsets of the constraints which are checked depending on what the user is trying to achieve.

User-action Dependent Checking

We can also ask the reverse: is it possible to reduce the amount of constraints that need to be checked depending on the user's actions? This idea was already mentioned in [AF00]. In the subset of J^3D_L without integrated Cones, Walls and Cubes the answer is definitely *yes*. If we consider a 3D Universe populated with Cone Trees it is certainly possible to move a complete Cone Tree without having to check any constraint regarding the

relative position of boxes. It would also be possible to remove the pipe between the two boxes without violating any constraint.

In the J^3D_L with integrated visualization techniques, it is much more complicated to find user actions that do not possibly conflict with anything. Any moving of an entity can be problematic, because it might lead to some entities of a structure (e.g a Cone) being inside and some outside of a Cube. Also, with the ghost boxes and the numerous restrictions, any moving might change the validity with which some structure resides inside a certain Cube.

11.4 Conclusion

We have seen in this chapter that different applications scenarios are possible. Depending on the scenario the role of certain constraints change. Some constraints are not checked as constraints in some scenarios, but are used to compute other relations, and sometimes constraints that are computed by external tools in one scenario are derived in others. As a result, the implementation must take the desired scenario into account when designing an editor. The scenario that fits the current shape of J^3D_L best is the Scene-to-Code scenario, where the ASG relations can completely be computed by the SRG ones.

Then we have described a possible sequence of activities to turn a scene into code and identified different components that participate in this process. Two of them are performing a mapping between the different "worlds", the scene parser connects the editor with the relation engine and the ASG parser separates the relation engine from the Java code. This way it is possible to keep the implementation of the connected components separated from each other.

Finally we have looked into different aspects regarding the amount of constraints necessary to check. One finding was that the editor might have features that render constraints useless. Whenever an editor is implemented in a way that rules out situations that are also forbidden by constraints it is no necessary to check the constraints. Reversely, we have concluded that if an editor wants to support J^3D_L well it has to implement features that reduce the amount of constraints necessary to check.

It might also be possible to reduce the number of constraints by focussing on the output the user wants to achieve. The constraints necessary to verify the correctness of the modelled Java structures constitute a different subset of J^3D_L than those to verify the "visual correctness" of Cone Trees, for example. By breaking up the complete conformity with J^3D_L into different semantically meaningful subgoals, the number of constraints to be checked can be reduced.

Benefits of the Declarative Approach

At this point the declarative approach of language specification shows its benefits. The relations and constraints can be specified using strings, as discussed in Sect. 11.2.3. Therefore, it is possible to integrate the constraints within the application very loosely, for example in a plain text file. That way the amount of constraints can be easily changed without having to modify the source code of the editor itself. The part of the application that controls the constraint checking can thus be implemented very generically.

With a very generic front-end that allows us to move entities in 3D, we could create modeling editors for numerous application domains simply by specifying a different subset of the J^3D_L constraints. If we want to create an editor for file system hierarchies, for example, we can tailor an editor out of the Information Cubes constraints of J^3D_L and by interpreting the ASG relations as modeling "files" and "folders" instead of "classes" and "packages". The only features the front-end must have are those for manipulating basic 3D objects and the accessability of the scene by a scene parser that turns the scene into the SRG relations.

On the other hand, a generic approach would oppose our philosophy of "constraints are requirements" for the editor. If the front-end is too generic and "unaware" of the constraints, the usage of such an editor might be a frustrating experience. If the user must correctly align a Cone Tree "by hand", for example, and gets feedback if the alignment is correct earliest when the whole scene is parsed, the acceptance of the tool would be low.

Finding out the right balance between *generic* and *specific* will be a very interesting route of further research. The declarative approach supports any combination of both.

Chapter 12

Summary

We have completed our task of defining the language in all its detail and we also looked into general considerations regarding implementation in the previous chapter.

We created the language in three steps. The first step was to create the relational "infrastructure", following the idea of [RS97] to describe the scene by different layers. In the second step, we went through the three visualization techniques individually. Not surprisingly, many the structural constraints for Cone Trees and Information Walls were the same as they covered the same concept: *inheritance*. The third step was the integration of the techniques which succeeded, but left us with open questions we will discuss further in our summary.

At this point, we want to give an evaluation following the initial goal outline of Chapter 3. First we will evaluate the approach in Sect. 12.1, then we discuss in Sect. 12.2, how the relationship between an editor and the language can be seen in the context of J^3D_L, and finally we sum up our experiences and insights we made with the visualization techniques in Sect. 12.3.

12.1 The Approach

All together we can say that the definition of a visual language through relation-algebraic constraints was successful. Helm and Marriott [HM91] give criteria that they think a visual language (picture specificational languages in their case) should possess. We want to begin the evaluation of our work with these criteria.

Hierarchical Specification

The first criterion is the ability to specify the language hierarchically by composition. We did that several times, most prominently when assembling

Cone Trees out of Cones. On a smaller degree, it was easily possible to modularize the language description by creating new relations. The fact that a box is connected to a pipe which is in turn connected to another box is originally expressible by two relations (pipeStarts, pipeEnds) which we encapsulated into the connectedByConePipe relation.

Natural Expressability

The second criterion of Helm & Marriott is that the language constructs have to be naturally expressible. Most of the relationships between (composed) entities were expressible using basic relational concepts such as univalence or totality. In the other cases, the constraints were relatively short. It was possible to find appropriate names for the concepts expressed by the relations, so that the translation from natural language into predicate logical statements was comprehensible. In this context, it must also be mentioned that the idea to adopt the Rekers & Schürr approach was a big help in terms of comprehensibility, as it nicely separated the different aspects of the language.

Following the ASG/SRG approach combined with relation algebra, we decided to limit our description capabilities to qualitative descriptions rather then having real-number arithmetics available. For all but two cases this limitation was appropriate. The Cone Trees were the spatially most complex structures we had to describe and we succeeded in doing so relatively easy. The only exception was the *Cone Shell*: we needed to ensure that Cones cannot overlap and to determine when a Cone Tree is inside a Cube. Here we needed to state a relation that creates a wrapper around the Cone for each Cone Tree. This entity is not under direct control of the user but a mere auxiliary construct that is not derived out of any other relations. At this point, the strict separation of language vs. editor was undermined, the editor appeared as the *deus ex machina* that solves an urgent problem of the language.

Operational Semantics

The third of the Helm & Marriott requirements is an efficiently implementable operational semantic. In the previous chapter, we discussed the implementation and the different scenarios. Due to RelView, the necessary work to implement a language compiler is equivalent to typing the constraints into a text file and feeding it to the component of an editor which deals with verification of a scene. Again, the Rekers & Schürr idea to clearly separate the spatial issues from the semantical ones showed its' benefit, as we were able to clearly define the points where the ASG and SRG were connected. We could easily and precisely define how the two layers must correspond, which in return could be interpreted as a formula to

compute the ASG out of the SRG. The reverse, creation of a scene out of source code, is not completely possible due to the "gap" between SRG and Physical Layout, that exists as we cannot derive the latter out of the first.

Next to these general criteria we must evaluate the aspects that are specific for our approach, especially the development of constraints out of predicate logical formulas.

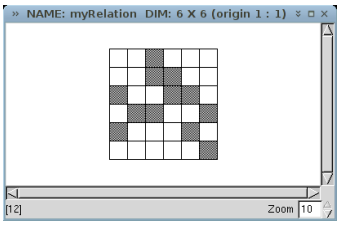
From Predicate Logic to Relational Algebra

It is harder to evaluate the possible benefits of transforming the constraints from predicate logic into relational algebra at this stage of the language development. From the practical point of view, it was of course necessary to do this translation, as our tool of choice (RelView) is a relation-algebraic system which needs relation-algebraic expressions. In this state of the research, we are not able to see the benefits of relation algebra on the conceptional side, though. Only in a few instances have we used the relation-algebraic calculus to prove properties of newly created relations. The benefit might become observable when turning to prove aspects like soundness and completeness of the language rigidly instead of only arguing for them. This was not possible to frame within the dimensions of one Masters' Thesis.

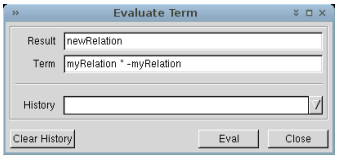
Creating an appropriate set of relation-algebraic constraints to describe the graphical structures was a long and complicated task with many failed attempts not documented by the current text of the thesis. This was especially true whenever the subcomponent (e.g. Cones of a Cone Tree) had to be represented and accessed later on (e.g. for comparison of their height). At times the decision for the relation-algebraic approach felt like a burden. Luckily, we found solutions but the initial learning curve was steep. Another source of pain was the manual transformation of predicate logic statements into relation-algebraic expressions. Most of the transformations were using only the most basic concepts such as union, intersection and composition. While typesetting the thesis in LaTeX it became obvious how much "copy & paste" followed by string replacement was applicable to the translation process. Software support for this task is highly desirable even if it is not fully automatized.

On the positive side (and as an successful example for tool support) the availability of RelView must be mentioned: not only does it offer a *backend* for evaluation of the relation-algebraic expressions but it also has a *frontend* with which example relations can be designed and relation-algebraic expression can be executed on it. Fig. 12.1 on the following page shows the most important windows of the RelView GUI.

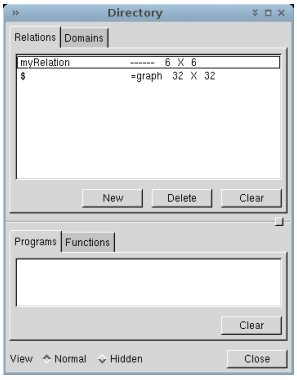
It allowed us to "play" around with constraints during their development and served as a "plausibility check" for constraints. In some cases it was even possible to modify the transformation of an erroneous predicate logical statement directly in RelView. Constraints (or parts of them) could be



(a) Editing a relation using the matrix view.



(b) Evaluting an expression in the evaluation window.



(c) A list of the defined relations in the directory window.

Figure 12.1: Parts of the RelView Graphical User Interface.

evaluated, and the visual feedback that the matrix visualization of such an output gave us was also often a source to develop an intuition on how a constraint "works".

12.2 The Relation Between Editor and Language

With the availability of J^3D_L, we have an implementation independent description of the collection of visualization methods and visual design decision developed in different places throughout the Vise3D project. This is, of course, a benefit in itself. Since we listed the constraints, we have a complete description of the language independently from the mathematical formalization even if we discovered that we do not need to check some of them. In particular, it allows us to discuss the relationship between the editor and the language.

As the constraints are basically strings specifying relation-algebraic expression we are able to modify J^3D_L easily. If only the Cone Trees are of interest the other constraints are simply dropped, if some constraints are found to be too restrictive they easily can be modified. From this point of view a highly generic editor might appear as the right solution. It would be able to support many variants of J^3D_L by depending fully on the validation component that checks these variants. Such an arrangement might be very

useful for doing experimental research on which concepts are most helpful for the user.

Looking at the different constraints, especially the geometrical ones, it appears questionable if a generic editor is the best solution, though. The intended use of an editor that bases on the J^3D_L is the development and exploration of software. If an editor is too generic it leaves the user with the burden of meeting the expectations of the constraints. The user would have to, for example, guarantee the correct alignment of the child boxes of a Cone and the correct positioning of the root box above the the center of the Cone's base circle. To perform these things in the required accuracy with a normal pointer device (mouse) seems to be an enervating and time consuming occupation which would absorb the benefits the 3D software engineering was introduced for in the first place. Therefore, it is likely to assume that an editor is equipped with a lot of functionality that guarantees a valid J^3D_L scene. The easy "configurability" of the language pays off in such an situation.

12.3 Insights on the Visualization Techniques

During the creation of the language we gained some insights on the visualization techniques themselves that we want to report on as well. First is the obvious similarity between Cone Trees and Information Walls. Both were used in J^3D_L to display inheritance structures. On first sight they look different (in terms of their visual appearance), as one is a tree where the other is a graph, but this difference faded when focusing on their description by constraints. The core of both descriptions was that the individual entities were connected by pipes. Two entities were part of the same Cone Tree or Wall only if they were connected. The way that the constraints allowed spheres, respectively boxes, to connect with each other created the difference between the tree structure and the DAG structure. Beyond this, it was also the basis for all geometrical considerations; boxes and spheres that were directly or indirectly connected had to meet certain geometrical constraints. This point of view might enable us to describe other visualization techniques as mere variations of the "connected by pipe" scheme.

The other noteworthy insight was made regarding the idea to integrate the three views into one and the idea to introduce ghost boxes/spheres to solve the problems it created. The fact that the Cube is an entity that *contains* the elements it is related to made it necessary to introduce ghost boxes and ghost spheres when integrating the Cubes with Cone Trees and Walls. The beauty of the ghost box is that it works as a "real" box in inside the Cone Tree but it shows at the same time that it does not belong there with the same right a solid box does. The problem with the inheritance and package membership is that they do not necessarily correlate. Being

part of the same package does not increase the probability of inheriting from each other. The rule was to include whole Cone Trees inside a Cube, even if only one box of the Cone Tree belongs there and the result is that a Cube might contain more boxes that do not belong there than those who do. It became worse when the display *implements* relationships added more reasons to include ghost entities. The statement we can make here from the conceptual analysis of the integration of the different views is that either the integration of Cone Trees and Cubes or the visualizations of packages as Cubes must be dropped. While this is a negative outcome, we consider it positive that our work was able to reveal this.

Chapter 13

Outlook

Not all insights that came up in the course of the thesis could have been dealt with, and as the summary shows, some aspects of the language require changes. Before concluding the thesis we want to point out potentially interesting tasks that could be the subject of further research in the context of J^3D_L.

An obvious progression of this thesis is the extension of J^3D_L. We have dealt with the core of object-orientation, classes, interfaces and packages but still other important concepts such as association, methods and attributes as well as their visibility are missing. It might be interesting to develop a relational description of object-oriented languages (at least Java) that is complete and independent from specific usage scenarios (e.g. 3D visualization or finding design patterns). In [BF04], applications are sketched out that would benefit from such an unified model. On the SRG level, the inclusion of more pipes is straightforward and we feel that with the relation-algebraic adoption of the ASG/SRG approach we have layed grounds even for easy addition of completely new visualization techniques.

With the outcome that ghost boxes and spheres are not the ideal solution for the Cube-Cone Tree integration, it seems preferable to think of new ways of integrating the different views. As we have argued, the problem seems to be that the package membership is visualized via containment and that this is problematic when integrating, for example, complete Cone Trees as these have to be contained completely. An easy way around this would be to render the package as Cubes but to express the membership relation by a special kind of pipe. As this might create other problems, further exploration seems necessary.

More generally it would be interesting to develop a better feeling for the editor-language relation. As we have argued, the user does not want to ensure all geometrical constraints by "hand" (by mouse). With the constraints readily available, it is now possible to examine where the line between editor support and language constraint should run.

On a more theoretical level, the language could be anchored more solidly in the the relational algebra when further language properties such as soundness and completeness are tackled depending solely on relation algebra. The fundamental relations with their asserted properties on both sides, ASG and SRG, would allow an axiomatization and a more rigid development of constraints and relations out of these axioms with formal proofs.

In order to better evaluate the pros and cons of the chosen approach, it could be helpful to start to model aspects of the language using a different formalism. An obvious candidate would be the Graph Transformation as it was the formalism of choice in [RS97], and therefore it would be able to also use the ASG/SRG separation.

With this paragraph we will conclude the thesis. As the summary in Chapter 12 has shown the approach is promising, yet numerous questions came up that point towards further research. We believe that this thesis was a contribution to substantiate the vision of Berghammer & Fronk in applying relational algebra to problems of visual languages and in a wider sense, to software engineering, and hope that others feel inspired to do so as well.

Bibliography

[AF00] K. Alfert and A. Fronk. 3-Dimensional Visualization of Java Class Relations. In M.M Tanik and A. Ertas, editors, *Proceedings of the 5th World Conference on Integrated Design Process Technology*. Society for Design and Process Science, 2000.

[AF02] K. Alfert and A. Fronk. Manipulation of 3-Dimensional Visualization of Java Class Relations. In *Proceedings of the 6th World Conference on Integrated Design Process Technology*. Society for Design and Process Science, 2002.

[Beh98] Ralf Behnke. *Transformationelle Programmenticklung im Rahmen relationaler und sequentieller Algebren*. PhD thesis, Technische Fakultät der Universität Kiel, Juni 1998.

[Ber03] Rudolf Berghammer. *Verbands- und Relationentheorie mit Anwendungen in der Informatik*. lecture notes, available at http://www.informatik.uni-kiel.de/~rub/SkriptVerbRel.ps.gz, Winter Term 2003/03. last visited April 20th, 2005.

[BF03] R. Berghammer and A. Fronk. Applying Relational Algebra in 3D Graphical Software Design. In *Proceedings 7th International Seminar on Relational Methods in Computer Science*, pages 89–96. Malente, May 2003.

[BF04] R. Berghammer and A. Fronk. Considering Design Problems in OO-Software Engineering with Relations and Relation-based Tools. *Journal on Relational Methods in Computer Science*, 1:73–92, December 2004.

[BKS97] C. Brink, W. Kahl, and G. Schmidt, editors. *Relational Methods in Computer Science*. Springer-Verlag, 1997.

[BS97] R. Berghammer and P. Schneider. Machine Support of Relational Computations: The Kiel RELVIEW System. Technical Report 9711, Institut für Informatik und Praktische Mathematik, Preusserstraße 1-9, 24105 Kiel, Germany, 1997.

[BSMM01] I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Verlag Harry Deutsch, fifth edition, 2001.

[DFSH04] E.-E. Doberkat, A. Fronk, and J. Schröder (Hrsg.). Abschlussbericht der PG 444 - Eclipse framework for editing complex three-dimensional software visualizations. SWT-Memo No 150, Software Technology, University of Dortmund, 2004. In german.

[Dwy01] Tim Dwyer. Three dimensional UML using force directed layout. In Peter Eades and Tim Pattison, editors, *Australian Symposium on Information Visualisation, (invis.au 2001)*, Sydney, Australia, 2001. ACS.

[Ecl] Eclipse project homepage. website, http://www.eclipse.org,. last visitied April 20th, 2005.

[Eng00] Frank Engelen. Conception and implementation of a three-dimensional class browser for java. Master's thesis, Software Technology, University of Dortmund, 2000. In german.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - elements of reusable object-oriented software*. Addison-Wesley, 1994.

[GJS97] J. Gosling, B. Joy, and G. Steele. *Java ^{TM} Die Sprachspezifikation*. Addison-Wesley, 1997.

[GRR99] M. Gogolla, O. Radfelder, and M. Richters. Towards three-dimensional animation of UML diagrams. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 489–502. Springer, 1999.

[HM91] R. Helm and K. Marriott. A declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing*, 2, 1991.

[Ja3] Java3D API. website, http://java.sun.com/products/java-media/3D. last visitied April 20th, 2005.

[KHP05] Kure - kiel university relation package homepage. website, http://www.informatik.uni-kiel.de/~progsys/relview/kure, last visitied April 20th, 2005.

[Mad91] R. D. Maddux. The origin of relation algebras in the development and axiomatization of the calculus of relations. *Studia Logica*, 50(3/4):421–455, 1991.

[MRC91] J. D. Mackinlay, G.G. Robertson, and S. K. Card. The perspective wall: detail and context smoothly integrated. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 173–179, New Orleans, Louisiana, United States, 1991. ACM Press, New York.

[OMG03] The Object Management Group. OMG unified modelling specification. online puplication, http://www.omg.org/technology/documents/formal/uml.htm, March 2003. last visited April 20th, 2005.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RG93] J. Rekimoto and M. Green. The Information Cube: Using transparency in 3d information visualization. In *Proceedings of the Third Annual Workshop on Information Technologies & Systems (WITS'93)*, pages 125–132, 1993.

[RHP05] Relview system homepage. website, http://www.informatik.uni-kiel.de/~progsys/relview/, last visitied April 20th, 2005.

[RMC91] G.G. Robertson, J.D. Mackinlay, and S. K. Card. Cone Trees: animated 3D visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194, New Orleans, Louisiana, United States, 1991. ACM Press, New York.

[Roh04] Oliver Rohr. Conception and implementation of three-dimensional visualization methods for exploration of object oriented software systems. Master's thesis, Software Technology, University of Dortmund, 2004. In german.

[RS97] J. Rekers and A. Schürr. Defining and Parsing Visual Languages with Layered Graph Grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[SS93] G. Schmidt and T. Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1993.

[Szy03] Oliver Szymanski. Relationale Algebra im dreidimensionalen Software-Entwurf - ein werkzeugbasierter ansatz. Master's thesis, Software Technology, University of Dortmund, 2003.

[Tch04] Jin-Ha Tchoe. Modification of three-dimensional Visualizations of object-oriented Software Structures. Master's thesis, Software Technology, University of Dortmund, 2004. In german.

[V3D] Vise3D homepage, project description. website, http://ls10-www.cs.uni-dortmund.de/vise3d/project.html. last visitied April 20th, 2005.

[WHF93] C. Ware, D. Hui, and G. Franck. Visualizing object oriented software in three dimensions. In *Proc. IBM Centre for Advanced Studies Conf., CASCON*, 1993.

[You96] P. Young. Three Dimensional Information Visualisation. Technical Report 12, Department of Computer Science, University of Durham, November 1996.

Erklärung

Name, Vorname: **Blom, Sören**

Hiermit erkläre ich, dass ich die Diplomarbeit mit dem Titel:

J^3D_L: Formalizing a Visual Language for Java Class Relations by Means of Relation Algebra

selbständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe.

Dortmund, den 22. April 2005 .....

(Unterschrift)