Lehmann,
Scoping Constraints and Reactive Behavior

Scoping Constraints and Reactive Behavior

Towards a Practical Object Constraint Programming Tool

Dynamische Constraints und Reaktives Verhalten

Werkzeuge zur Praktischen Anwendung von Objektconstraintprogrammierung

by

Stefan Lehmann

A thesis submitted to the
Hasso-Plattner-Institute
at the University of Potsdam, Germany
in partial fulfillment of the requirements for the degree of

Master of Science in IT-Systems Engineering

Supervisors

Prof. Dr. Robert Hirschfeld
Tim Felgentreff, M.Sc.

Software Architecture Group
Hasso-Plattner-Institute
University of Potsdam, Germany

April 30, 2015

Abstract

Constraints allow developers to specify desired properties of a program in a declarative manner. These properties are solved and maintained automatically by specialized constraint solvers. This results in clean, compact code, and avoids scattered code fragments to imperatively (re-)satisfy desired properties. Despite its advantages, constraint programming is not widespread, with imperative programming still the norm.

The object constraint programming paradigm attempts to bring constraints to a wider audience by cleanly integrating constraints with object-oriented languages. Object constraint programming languages unify the constructs for encapsulation and abstraction by using object-oriented method definitions for both declarative and imperative code. The Babelsberg design is a concrete instance of the object constraint programming paradigm and targets object-oriented programmers who are typically not familiar with constraints. One major goal of Babelsberg is to provide constraints as a practical tool for imperative programmers.

We evaluate how well Babelsberg fulfills this goal with an object-oriented application: an interactive online game. We implement this application in JavaScript and use Babelsberg constraints in appropriate places. We show where constraints can improve the readability and understandability of object-oriented programs. However, we also identify three major shortcomings in the design of Babelsberg with regards to object-oriented applications. First, some constraints involving high-level objects cannot be solved by any available solver. The attempt to solve such a constraint may result in inconsistent system states. Second, in contrast to object-oriented programming, constraint programming strictly separates state and behavior, e.g. by hiding the solver logic from the programmer. Third, Babelsberg only provides primitive operations to dynamically enable or disable constraints, yet object-oriented environments demand frequent changes and adaptability.

To overcome these shortcomings, we refine the current Babelsberg design in three ways. Regarding the first issue, we propose a concept that reverts the program to the last valid state in case of a breaking constraint. To address the second issue, we introduce *reactive constraints* that allow to invoke or adapt user-defined behavior once or as long as a condition evaluates to true, respectively. To tackle the third issue, we treat the context explicitly. All constraints associated with a context are automatically enabled as long as the program remains in this context. Activating the context based on arbitrary boolean expressions results in a convenient scoping mechanism.

We implement the proposed concepts and illustrate their usage in our example application. We argue that the proposed concepts represent a useful addition to Babelsberg and to the object constraint programming paradigm in general.

Zusammenfassung

Mit Hilfe von Constraints können Entwickler gewünschte Eigenschaften eines Programmes auf deklarative Weise beschreiben. Diese Eigenschaften werden von spezialisierten Constraintlösern herbeigeführt und automatisch aufrechterhalten. Dies resultiert in klarem und kompaktem Quelltext und verhindert verstreute Quelltextfragmente, die die gewünschten Eigenschaften imperativ herbeiführen. Trotz ihrer Vorteile ist die Constraintprogrammierung nicht weit verbreitet.

Das Ziel der Objektconstraintprogrammierung ist es, den Einsatz von Constraints leichter zu ermöglichen, indem diese in objektorientierte Sprachen integriert werden. Hierzu vereinheitlich die Objektconstraintprogrammierung die Konstrukte für Kapselung und Abstraktion, indem sie objektorientierte Methodendefinitionen sowohl für deklarativen als auch imperativen Quelltext verwendet. Ein Beispiel für eine Sprache der Objektconstraintprogrammierung ist Babelsberg. Babelsberg richtet sich an objektorientierte Programmierer, die typischerweise nicht mit Constraintprogrammierung vertraut sind. Für diese Zielgruppe möchte Babelsberg Constraints als praxistaugliches Werkzeug bereitstellen.

Am Beispiel einer objektorientierten Anwendung untersuchen wir, ob Babelsberg dieses Ziel erreicht hat. Wir implementieren diese Anwendung in JavaScript und nutzen Babelsberg an geeigneten Stellen. Anhand der Implementierung zeigen wir, dass Constraints die Lesbarkeit objektorientierter Programme verbessern können. Jedoch erkennen wir auch drei Probleme Babelsbergs im Hinblick auf objektorientierte Anwendungen. Erstens, einige Constraints, die sich auf komplexe Objekte beziehen, können von keinem verfügbaren Constraintlöser gelöst werden. Der Versuch diese Constraints zu lösen könnte in inkonsistenten Systemzuständen resultieren. Zweitens, im Gegensatz zu objektorientierter Programmierung trennt die Constraintprogrammierung Zustand und Verhalten strikt voneinander, beispielsweise indem sie die Logik des Constraintlösers vor dem Programmierer verbirgt. Drittens, Babelsberg stellt lediglich einfache Operationen zur dynamischen Aktivierung von Constraints bereit. Objektorientierte Systeme hingegen verlangen häufige Änderungen und Anpassungsfähigkeit.

Um diese Probleme zu überwinden, erweitern wir Babelsberg in dreierlei Hinsicht. Das erste Problem kann teilweise gelöst werden, indem im Falle eines fehlschlagenden Costraints das System wieder in den letzten gültigen Zustand zurückgesetzt wird. Auf das zweite Problem eingehend führen wir *reaktive Constraints* ein. Reaktive Constraints erlauben es nutzerdefiniertes Verhalten auszulösen oder zu verändern, sobald oder solange eine Bedingung gilt. Für das dritte Problem nutzen wir ein explizites Kontextobjekt. Alle mit diesem Kontextobjekt assoziierten Constraints sind solange aktiv wie sich das Programm in diesem Kontext befindet. Kombiniert mit der Aktivierung der Kontextobjekte basierend auf beliebigen booleschen Ausdrücken steht somit ein geeigneter Aktivierungmechanismus für Constraints zur Verfügung.

Wir implementieren die vorgeschlagenen Konzepte in JavaScript und zeigen deren Nutzung anhand unserer Beispielanwendung. Wir sind der Meinung, dass die vorgeschlagenen Konzepte eine nützliche Ergänzung zu Babelsberg und Objektconstraintprogrammierung im Allgemeinen darstellen.

Acknowledgments

First of all, I want to thank Robert Hirschfeld and Tim Felgentreff for the opportunity to work with the Software Architecture Group. I have greatly benefited from your advice and support. Thanks for the numerous discussions about the design and implementation of the proposed concepts.

I would like to express my gratitude to all members of the group for the constructive comments on my work and a pleasant working atmosphere.

Special thanks to Maria Graber for help to get things up and running as well as lots of puzzles to solve.

I want to thank Sharon Nemeth for the immediate answers to all my questions regarding English grammar and scientific writing.

Many thanks to Cathleen Ramson, who always listened to my problems. Thank you for always cheering me up the whole time.

I appreciate the feedback offered by all proofreaders. You encouraged me to continuously improve this thesis.

My deepest appreciation goes to all my friends for making stressful times enjoyable. Together, we regained our strength through strategic board games and exhausting hiking tours.

Finally, I want to thank my family for always supporting my dreams and goals.

Contents

List of Figures

List of Listings

List of Abbreviations

AOP	aspect-oriented programming
CIP	constraint imperative programming
CLP	constraint logic programming
COP	context-oriented programming
CP	constraint programming
CSP	constraint satisfaction problem
GUI	graphical user interface
OCP	object constraint programming
OO	object-oriented
OOP	object-oriented programming

1 Introduction

Constraints and the constraint programming (CP) paradigm occur in a variety of application domains, including graphics, CAD/CAM, and planning. A constraint is a desired property of a program that should hold, and usually represents a state invariant. An important property of constraints is that they are declarative. Constraints allow the programmer to specify *what* should be the case rather than *how* to achieve it. They are solved and maintained automatically by constraint solvers. Constraint solvers typically work on a limited domain and encapsulate the complexity required to solve for desired properties. This results in compact, clean code, and avoids scattered code fragments to imperatively (re-)satisfy the desired property. Despite its advantages and long history of research, CP is not widespread, with imperative programming still the norm.

1.1 Babelsberg, a Practical Constraint Programming Tool for Object-Oriented Developers?

The object constraint programming (OCP) paradigm attempts to bring constraints to a wider audience by cleanly integrating constraints with object-oriented (OO) languages. OCP languages unify the constructs for encapsulation and abstraction by using object-oriented method definitions for both declarative and imperative code. This allows imperative programmers to employ constraints using familiar OO constructs.

A concrete instance of the OCP paradigm that this thesis explores is Babelsberg [11]. Babelsberg is targeted at OO programmers who are typically not familiar with constraints and CP. One major goal of Babelsberg is to provide constraints as a practical tool for imperative programmers.

In this thesis, we want to identify to what degree Babelsberg already achieved this goal. To do so, we evaluate a non-trivial application scenario which is implemented using Babelsberg in appropriate places. The Babelsberg design is implemented in multiple languages, including Ruby [11], JavaScript [12], Smalltalk [19], and Python. Among those implementations Babelsberg/JS is the implementation that is most advanced and conforming with the Babelsberg standard specification [13]. Therefore, we chose Babelsberg/JS for the provided sample application. While implementing the application, we discover three major shortcomings of Babelsberg with regards to OO applications. First, some constraints involving high-level objects cannot be solved by any available solver. The attempt to solve such a constraint may result in inconsistent system states. Thus, programmers still need to be aware of the limitations of constraint solvers. Second, Babelsberg and CP languages in general focus on state, not on behavior. As a consequence, it is not possible to invoke or adapt user-defined behavior based on constraints. Thus, programmers can only describe desired state, but no desired behavior, using Babelsberg. Third, constraints are enabled and disabled more frequently in OO applications

than in traditional CP domains. However, Babelsberg only provides primitive operations to dynamically adapt constraints.

To overcome these shortcomings, this thesis presents refinements to the current Babelsberg design. To address the first issue, unsolvable constraints, we propose a concept that reverts the program to the last valid state in case of a breaking constraint. To tackle the second issue, the missing integration of constraints with behavior, we introduce *reactive constraints*, i.e. constraints that adapt or invoke user-defined behavior once or as long as a condition evaluates to true, respectively. Regarding the third issue, the need for a powerful scoping mechanism, we treat the context explicitly. All constraints associated with a context are automatically enabled as long as the program remains in this context. We then show that these additions are useful not only in the provided example, but also for OCP languages in general.

1.2 Contributions and Structure of this Thesis

As mentioned before, the goal of this thesis is to move Babelsberg into the direction of a practically usable OCP language. The contributions are:

- A description of the shortcomings of the current Babelsberg design with respect to OO applications.
- The design of *reactive constraints* that integrate constraints with OO behavior.
- An extension of Babelsberg with a convenient scoping mechanism based on explicit context objects.
- The implementation of the designed concepts in Babelsberg/JS.
- A non-trivial application using our extensions and OO constructs in conjunction.
- A discussion of the concepts in comparison with alternative implementation strategies.

In the remainder of this thesis, Chapter 2 covers relevant background such as key concepts and major systems of CP. Chapter 3 surveys several examples created using Babelsberg, of these all originate from classical CP domains. Chapter 4 provides a non-trivial, OO application scenario and identifies the shortcomings of Babelsberg regarding this example. Concepts to overcome these shortcomings are presented in Chapter 5. An implementation of these concepts is explained in Chapter 6. Chapter 7 evaluates the concepts in comparison with alternative implementation strategies. Related concepts are presented in Chapter 8. Future work and the conclusions are described in Chapter 9 and Chapter 10, respectively.

2 Background

This chapter reviews the key concepts of constraint programming (CP) with a particular focus on constraint imperative programming (CIP) and object constraint programming (OCP). We discuss major CP systems as well as recent efforts to integrate CP into existing languages, with a focus on the Babelsberg design. Finally, we present a brief overview of the constraint solvers available to Babelsberg/JS.

2.1 An Incomplete History of Constraint Programming

The earliest notable example of a CP system is Sutherland's Sketchpad [37] that demonstrates three of the major features of CP, as described by Wallace [39]: declarative problem modeling, efficient search for feasible solutions, and propagation of effects of decisions. Sketchpad allows to define geometric shapes through a graphical user interface (GUI). The first major feature of CP is declarative problem modeling. The programmer can describe the desired system state and, thereby, abstract from how this state is accomplished. For instance, Sketchpad enables the user to describe figures declaratively using a number of predefined constraints such as equal length or parallelism of lines. The usage of constraints allows the definition of more complex shapes. The second feature of CP is the efficient search for a feasible solution. To do so, Sketchpad supports three different solvers, including propagation of degrees of freedom, a local propagation solver, and an iterative relaxation solver. The last two are discussed in more detail in Section 2.3. If the preferred solver fails, Sketchpad falls back to one of the other solvers. The third feature of CP is the propagation of effects of decisions: when a constraint system is modified from the outside, this modification is propagated through the constraint solver. For instance, in Sketchpad the user can modify constrained shapes. Then, this change is recognized by the system which in turn instructs the constraint solver to maintain the constraints.

The three features of CP lead to many early applications related to graphics, like geometric layouts or user interface toolkits. One such graphical simulation system is ThingLab [5]. ThingLab allows its users to define building blocks and relations using constraints for a given domain, e.g. a resistor that obeys Ohm´s law. Then, users employ these building blocks to construct complex simulations, like an electrical circuit using resistors, batteries, and wires, in a graphical manner. Several different systems could be built and simulated on the basis of constraints. ThingLab´s examples range from layouting to simulations of bridges under load. To solve the resulting constraint systems ThingLab relies on the same system of solvers as Sketchpad.

Another declarative programming approach, logic programming, caters to different domains such as simultaneous equations. The integration of constraints into logic programming languages create the powerful concept of constraint logic programming (CLP). One example is the CLP($\mathcal{R}$) language [23] that extends the logic programming language Prolog [9] with a constraint solver for real numbers. This integration opens a way to solve combinatorial prob-

lems involving simultaneous linear equations for simulations and finite-domain problems for logic puzzles. In consequence, most Prolog systems today contain one or more finite domain solvers as well as solvers specialized on a particular domain like real numbers.

Constraints over these domains cannot be expressed directly in imperative programming languages. As a result, the constraints are encoded implicitly and their continuous maintenance is ensured through scattered code fragments. To address this issue, the constraint imperative programming (CIP) design attempts to syntactically integrate constraints with object-oriented, imperative languages. Unlike CLP, CIP supports object-oriented (OO) concepts, including classes, mutable objects and object identity, that are familiar to imperative programmers.

Kaleidoscope [31] was one of the first languages to integrate constraints and OO concepts and coined the term constraint imperative programming (CIP). Built-in constraints in Kaleidoscope can only be specified over primitive objects. In addition, Kaleidoscope provides constraint constructors to specify user-defined constraints in terms of other user-defined constraint or primitive constraints. Constraints can be required or non-required, with multiple levels of preference. Non-required constraints should be satisfied only if they are not contradictory to required constraints. This is based on the theory of constraint hierarchies [6]. Additionally, the duration of constraints can vary from single time usage, over block-wide activation to constraints that are active for the remainder of the execution. The constraint model of Kaleidoscope changed over its years of development. Earlier versions, Kaleidoscope'90 [14] and Kaleidoscope'91 [15], are based on the model of constraints used in CLP. In this *refinement* model constraint variables are represented by their domain, and constraints add further restrictions to this domain. Later versions of Kaleidoscope implement the *pertubation* model [30]. In this model constraint variables refer to a single object. Destructive assignments invalidate existing constraints and in order to (re-)satisfy the constraints, the constraint solver changes the values of connected variables.

In contrast to Kaleidoscope, TURTLE [20] combines not only imperative concepts and constraints but functional programming concepts as well. TURTLE clearly separates ordinary variables from constraint variables by requiring the programmer to declare both types explicitly. Ordinary variables participate in a constraint only as constants. Similar to functions, TURTLE introduces *user-defined constraints* as a unit of reusable constraint statements. These user-defined constraints may be called from constraint expressions. If ordinary functions are called from constraints, their return value is used as a constant. Like Kaleidoscope, TURTLE supports required and non-required constraints. Constraints can be globally active, or scoped to the execution of a code block.

In contrast to Kaleidoscope and TURTLE which are languages themselves, Backtalk [35] is a library written in Smalltalk [18]. Backtalk is heavily influenced by CLP. It provides a finite-domain constraint satisfaction problem (CSP) solver that uses a combination of arc-consistency and backtracking techniques to reduce complexity. Constraint construction is syntactically integrated into Smalltalk, allowing arbitrary Smalltalk blocks as constraint expressions. The domains of constraint variables can contain any Smalltalk object. However, these constraint variables have to be constructed explicitly. Backtalk showed that the understanding of high-level objects can actually increase the performance of CSPs [33]. This is done by using a level-wise approach, solving constraints on primitive types first. Thereby, reducing the domain of higher-level objects, and ultimately improve the performance of high-level

constraints. Backtalk enables its users to model classical CSP systems using OO abstraction. Available examples include the n queens problem, timetabling variants, logic puzzles, and a case study on automatic harmonization [34].

To sum up, CP occurs in a variety of application fields, e.g. layouting, automatic time-tabling, and physics simulations. CIP provides an integration of constraints and imperative languages, allowing the usage of OO abstractions for constraint construction. However, CIP is no breakthrough, imperative programming and object-oriented programming (OOP) are still the norm. Nevertheless, CIP paves the way for more sophisticated integrations, as described in Section 2.2.

2.2 Recent Efforts in Constraint Programming

OCP picks up and extends the concepts of Kaleidoscope. An OCP language cleanly integrates constraints into the underlying host language. To do so, OCP languages unify the constructs for encapsulation and abstraction by only using OO method definitions for both declarative and imperative code. In contrast, many CIP approaches emphasize separate, parallel concepts to specify constraints, rather than reuse known OO definitions.

A concrete instance of the OCP paradigm is the Babelsberg design [11] which this thesis extends. The main goal of Babelsberg is to make constraints a useful tool for OO programmers. Therefore, Babelsberg provides the fewest number of new keywords, and can be seen as the smallest increment of existing OO paradigms. In contrast to Kaleidoscope, Babelsberg provides a much simpler approach. Babelsberg uses ordinary methods and standard method dispatch, rather than specialized constraint constructors and multi-method dispatch. However, a method used as a constraint expression is subject to a number of restrictions [13]:

1) Evaluating the constraint expression should return a boolean.
2) The expression that defines the constraint should be free of side effects.
3) Evaluating the constraint expression should be deterministic.

As an example of defining constraints, consider a rectangle object with some unknown internal structure. The rectangle object provides convenient accessor methods in its application programming interface, e.g. `width`, `height`, and `area`. We can make use of these methods to define constraints while respecting object encapsulation. Babelsberg does not only support simple accessor methods like in this case but also complex, calculated properties. Suppose we want the rectangle to have a minimum extent, so that it would be visible on screen. Listing 2.1 shows the definition of a constraint that restricts the extent of the rectangle in both dimensions to a certain threshold.

```
1 always: { rect.width() >= 10 && rect.height() >= 10 }
```

Listing 2.1: A simple constraint in Babelsberg/JS

In Babelsberg/JS [12], the `always` expression generates a call to a global constraint construction function with the given predicate. In Listing 2.1 and the following examples a source-to-source transformation is used to provide syntactic sugar. Details on the transformation are described by Section 6.1. The constraint expression can be any statement in the host language, in this case JavaScript, and may use imperative concepts, such as conditionals, loops, even recursive method calls.

Solving constraints generally is NP-hard, but for certain restricted yet useful classes of constraints, more efficient solvers are available. These solvers typically have several restrictions on what kind of constraints they can solve. In order to cover a wide range of applications with good performance, Babelsberg makes use of multiple specialized solvers. These solvers can interoperate to solve a particular problem using an architecture of cooperating solvers. Listing 2.2 shows an example for cooperating solvers. The first `always` expression is equivalent to the constraint in Listing 2.1, but this time we specify that it should be solved using the Cassowary constraint solver [4]. This solver is capable of solving linear problems over reals. The second constraint contains a non-linear expression in line 5. It cannot be solved by Cassowary, so, we ask the system to use the relaxation solver from Sutherland's Sketchpad to solve it. Both constraint solvers are discussed in Section 2.3.

```
1 always: { solver: cassowary
2   rect.width() >= 10 && rect.height() >= 10
3 }
4 always: { solver: sutherland
5   rect.area() >= 1000
6 }
```

Listing 2.2: Cooperating solvers in Babelsberg/JS

Babelsberg provides several additional features that make constraints useful in a wider area of domains in particular for imperative languages. First, in the presence of frequent changes typical in imperative environments, Babelsberg can use, but is not limited to incremental solvers. These solvers allow to efficiently (re-)satisfy a set of constraints as constraint variables are modified or constraints are added. Second, Babelsberg allows the definition of read-only variables. Variables marked as read-only cannot be changed by constraint solvers. The programmer can still modify them, possibly invalidate certain constraints, and, in turn, cause a constraint solver to (re-)satisfy these constraints. Third, Babelsberg supports soft constraints in addition to required constraints. In contrast to required constraints which have to be satisfied, soft constraints introduce several levels of preferences. A *weak* constraint will only be satisfied, if there is no contradictory constraint with a higher preference. This feature is useful for implicit stay constraints created by Babelsberg. Stay constraints are defined on concrete constraint variables. So, if possible, variable values do not change during constraint satisfaction. Without stay constraints solvers would simply search a fast but unstable solution. Using stay constraints allow complex systems to remain relatively stable, as expected by imperative programmers.

In Babelsberg, soft constraints can also be defined by the user. For example, in the domain of layouting, certain constraints are more favorable than others. In Listing 2.3 we define a constraint that is not required, but strongly preferred. We prefer that our rectangle is in portrait format, if possible.

```
1 always: { priority: 'strong'
2   rect.height() >= rect.width()
3 }
```

Listing 2.3: A soft constraint in Babelsberg/JS

In brief, Babelsberg aims to provide constraints as a useful tool for OO programmers. OO programmers can specify constraints in their chosen language using familiar OO concepts. Thereby, Babelsberg introduces the fewest number of new concepts to OO programmers.

2.3 On Available Constraint Solvers

The power of CP rests on the ability of the constraint solvers to solve and maintain desired properties. These solvers encapsulate the complexity of solving algorithms and provide a declarative interface to describe desired properties. However, most solvers are limited to specific tasks and domains in order to be efficient. Babelsberg/JS supports multiple solvers to cover a wide range of domains. These solvers are described in the following with their capabilities and configuration needs.

DeltaBlue. DeltaBlue is a local propagation solver that allows to specify multi-way constraints [16, 17]. In addition to the constraint expression, the programmer has to specify propagation functions for each possible way the constraint could be (re-)satisfied. So, unlike most other solvers DeltaBlue requires the user to specify how to solve desired constraints. This makes DeltaBlue harder to use, but also very powerful. However, Babelsberg/JS tackles the issue of explicitly specifying propagation functions. The current implementation of DeltaBlue in Babelsberg/JS provides a way to automatically conduct a one-way propagation function from the given constraint. The constraint variables can be arbitrary objects or primitives. DeltaBlue is an incremental solver that needs linear time during solving with respect to the number of constraints. When constraints are added or removed, DeltaBlue has to adjust its execution plan. This takes exponential time with respect to the number of constraints. As each assignment is modeled as a temporary constraint, one might want to use edit constraints for variables that frequently change. Edit constraints prepare the solver's internal structure for fast resolving when variables change. In case of DeltaBlue, edit constraints allow to mitigate the recalculation of the execution plan which assignments would normally trigger. DeltaBlue also supports stay constraints, so complex systems remain relatively stable in the presence of change.

Cassowary. The simplex solver Cassowary [4] can deal with linear arithmetic. The solver is restricted to real numbers and supports linear equations and non-strict inequalities, but not strict inequalities. Solving strict inequalities in the realm of real numbers requires the deviation from the optimal solution to become infinitely small. However, this is not supported in the binary representation as floats. Despite these restrictions, Cassowary is very efficient. As a result, Cassowary is mainly used in the domain of responsive user interfaces. Recent work using Cassowary includes the Mac OS X [36] layout specification language and the Python GUI framework Enaml[1]. Like DeltaBlue, Cassowary is an incremental solver. Cassowary needs exponential time during constraint construction, but only linear time during solving with respect to the number of constraints. Therefore, it is a common practice to use edit constraints for user interaction, e.g. a continuous resizing of the main window. That way, Cassowary's execution plan, the simplex tableau, is only calculated once for a sequence of mouse movements. Additionally, Cassowary supports mandatory and preferred constraints, creating a constraint

[1] Enthought Inc., Enaml 0.6.3 documentation, http://docs.enthought.com/enaml/ (last accessed April 30, 2015)

hierarchy. Stable solutions are especially important in the domain of layouting. Thus, Cassowary supports implicit stay constraints and is optimized for *weighted-sum-better* solutions to provide stable layouts. Unlike DeltaBlue, Cassowary can also handle cycles of constraints, which often occur in over-constrained layouts.

Z3. The Z3 theorem prover [8] is an SMT solver by Microsoft Research. An SMT solver's main purpose is to determine whether a given set of formulas is satisfiable. Z3 covers a wide range of domains, e.g. integers, real numbers, booleans, and even strings. Additionally, own data types can be defined, making Z3 suitable to solve typical CSP tasks. Z3 supports linear and non-linear arithmetic and boolean logic. Unlike Cassowary or DeltaBlue, Z3 is not optimized for incremental use. Each change, regardless whether a constraint is modified or an assignment occurs, requires a complete recalculation of the variable assignments. Recent versions of Z3 support soft constraints. This change allows stable solutions, and, in turn, Z3 can be used effectively in the context of imperative programming. Sample application fields of Z3 include logic puzzles, assignments, and scheduling tasks.

Sutherland's relaxation. Several relaxation algorithms have been proposed till today. Babelsberg/JS supports a concrete relaxation solver similar to the one used in Sketchpad [37]. Like Cassowary, this algorithm operates on real numbers, but unlike Cassowary, this solver can also solve non-linear equations and inequalities. To do so, the solver needs to be provided with some error functions. These error functions return a number that indicates how much the current solution deviates from an optimal solution, i.e. the constraint is satisfied. Using Babelsberg/JS, this error function can be automatically constructed given a constraint expression. Given the error functions, an iterative algorithm is used to (re-)satisfy the constraints. One by one each variable is adjusted according to the error functions. For each error function, the deviation with respect to the variable is taken. Then, the variable is adjusted a little in the direction of its deviation. This process is repeated for each variable until the system converges.

This method can easily deal with with over- and underconstrained systems. However, there are also a number of disadvantages. First, the algorithm does not employ soft constraints. Second, assignments are handled as ordinary constraints. Thus, the assigned variable will not always have the value assigned. This might be unexpected by imperative programmers. Third, it could take several iterations till the constraint system is satisfied. To deal with this, practical implementations employ either a timeout, like in Babelsberg/JS, or define an upper bound to the number of iterations. Hence, this can lead to imprecise solutions. And finally, the system even may diverge in presence of large input changes. Nevertheless, relaxation found potential usage in the areas of geometric drawing as well as physical simulations. In the latter case, the number of iterations is limited on purpose to simulate time-dependent behavior.

Finite-domain constraint satisfaction problem (CSP) solver. The final type of solver currently available in Babelsberg/JS is a finite-domain CSP solver similar to the one used in Backtalk. The solver allows arbitrary constraint expressions over constraint variables of arbitrary domains. However, these constraint variables and their domains must be declared explicitly. This is contrary to the idea of object encapsulation in OCP. Babelsberg/JS assumes certain default domains to known primitive data types if not declared otherwise.

Several algorithms to solve a CSP exist. Naïve backtracking typically results in a high runtime for problems with large domains. So, variants to ensure local consistency are desirable. Arc consistency can greatly reduce runtime by reducing the domain of variables participating in a constraint. Currently, this is only supported for constraints with 2 participating variables. Still, the runtime of a CSP highly depends on the concrete constraints and constraint variable domains.

Typically a CSP is solved by returning the first valid solution. No stay constraints are provided by default. Nevertheless, Babelsberg/JS achieves stable behavior by sorting the values of a domain by their distance to the current value. Thus, values with minor changes are selected first. The CSP solver provides no support for soft constraints with all constraints considered as required. Typical application fields for CSP solvers include logic puzzles and resource allocation.

3 Motivation

Babelsberg aims to provide constraints as a practical tool for object-oriented (OO) programmers. We survey the most important example applications created with Babelsberg/JS. We do this to ascertain to what degree Babelsberg already achieved this goal. We show several shortcomings with respect to the examples that make them inappropriate to verify our initial question.

3.1 Sample Applications

Babelsberg/JS features several example applications that originate from various domains. In order to analyze the capabilities of Babelsberg we take a look at these examples in the following.

Temperature converter. An interactive temperature converter is one of the classical examples of constraint programming (CP) systems. The ThingLab system, described in Section 2.1, provides an equivalent example. Among other sample applications[1] the temperature converter is built in the Lively Kernel self-supporting development environment [22]. Figure 3.1 shows a screenshot of the application. It consists of two sliders associated with a Celsius and a Fahrenheit value as well as labels that display the values. A user of the application can manipulate the temperature values via the sliders. Changing one value automatically updates all other values. To achieve this, the four components are connected via the constraints in Listing 3.1. These constraints ensure data consistency between the values. For instance, in lines 7 and 8 a Cassowary solver ensures that the handles of the `Sliders` are restricted to the visible range. Additionally, Cassowary also takes care of the conversion between the Celsius and Fahrenheit values as stated in line 9. This is possible because the conversion only involves linear equations which Cassowary is capable of solving. However, Cassowary is incapable of dealing with string values needed by the `Texts` to display the values. To solve constraints over these `Texts`, the temperature converter uses a second solver, DeltaBlue. Lines 13 and 14 ensure equality between the string value and the converted value of the respective `Slider`. Both solvers are able to interoperate through Babelberg's architecture of cooperating solvers. This is also the main showcase purpose of the example.

Fabric.js layouting. Layouts are easy to describe yet hard to maintain using imperative languages, making layouting a typical application field of CP. Babelsberg/JS was used to implement an interactive layouting demo using a fabric.js[2] canvas. In this example the user can manipulate layout components graphically, as seen in Figure 3.2. Additionally, the user can add

[1] Tim Felgentreff, Babelsberg/JS Demo, http://doi.org/10.1007/978-3-662-44202-9_17 (last accessed April 30, 2015)

[2] Fabric.js Javascript Canvas Library, http://fabricjs.com/ (last accessed April 30, 2015)

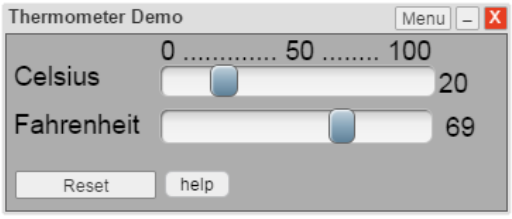


Figure 3.1: Interactive thermometer demo in LivelyKernel using Babelsberg/JS

```
var f = this.get('FahrenheitSlider'),
    c = this.get('CelsiusSlider'),
    fl = this.get('FahrenheitLabel'),
    cl = this.get('CelsiusLabel');

always: { solver: new ClSimplexSolver()
  f.getValue() >= 0 && c.getValue() >= 0 &&
  f.getValue() <= 1 && c.getValue() <= 1 &&
  (f.getValue() * 100) - 32 == (c.getValue() * 100) * 1.8
}

always: { solver: new DBPlanner()
  fl.getTextString() == Math.round(f.getValue() * 100).toString() &&
  cl.getTextString() == Math.round(c.getValue() * 100).toString()
}
```

Listing 3.1: Constraints in the interactive thermometer demo

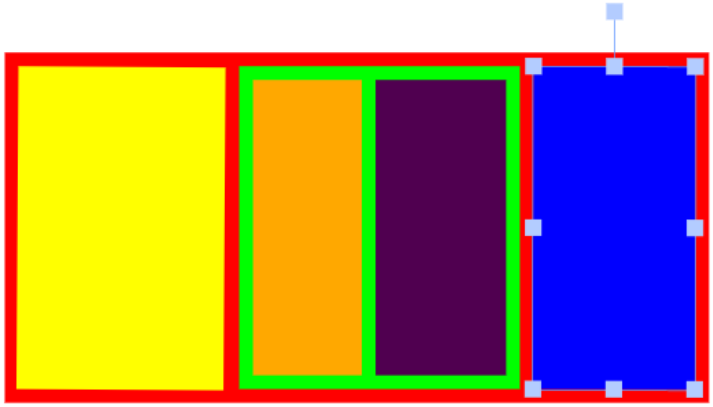


Figure 3.2: Interactive layouting application using fabric.js and Babelsberg/JS

constraints using an in-application code editor. When the code in the editor changes, the application removes all current constraints on the layout components. Then, the application creates new constraints by running the user-defined code. These constraints restrict the properties of the layout components. The user immediately sees the effect of the constraints, and can manipulate the resulting layout.

```
1  function horizontalBox(box, children, margin) {
2    function right(rect) {
3      return rect.left + rect.scaleX * bbb.readonly(rect.width);
4    }
5    function bottom(rect) {
6      return rect.top + rect.scaleY * bbb.readonly(rect.height);
7    }
8
9    always: { box.left + margin == children.first().left }
10
11   children.reduce(function(prev, current) {
12     always: { right(prev) + margin == current.left }
13     return current;
14   });
15
16   always: { right(box) == right(children.last()) + margin }
17
18   children.each(function(child) {
19     always: { child.top == box.top + margin }
20     always: { bottom(child) + margin == bottom(box) }
21   });
22 }
23
24 horizontalBox(this.red, [this.yellow, this.green, this.blue], 10);
25 horizontalBox(this.green, [this.orange, this.purple], 10);
```

Listing 3.2: Constraints to define a nested box layout

Listing 3.2 shows the constraints used to create the nested horizontal box layout in Figure 3.2. The main functionality of this example is provided by the `horizontalBox` function. This function gets an enclosing `Rect`, a list of child `Rects`, as well as the desired margin around the child `Rects`. It takes several constraints to create a horizontal box layout. First, in line 9 the left side of the first child and the enclosing box are declared to be equal considering the given margin. Next, we constrain each two subsequent child `Rects` in line 12. To be precise, the right side of the first `Rect` should be equal to the left side of the next child considering the margin. Fabric.js does not support a `right` attribute by default. Similar, fabric.js does not provide a `bottom` attribute. So, lines 2 to 7 declare convenient accessor methods to overcome this shortcoming. These methods simplify the access to the right and bottom coordinates needed to define a horizontal box layout. The `width` and `height` attributes are marked as read only using `bbb.readonly` to disallow solvers to change them. This is necessary, because fabric.js expect its users to not change these values. Later, the right side of the rightmost child is constrained to the right side of the enclosing box in line 16. This concludes the horizontal alignment of the child `Rects`. However, the vertical alignment is still missing. So, the top and bottom of each child are aligned to the enclosing box in lines 19 and 20. Now, in lines 24 and 25 we use the `horizontalBox` function to actually create a nested horizontal box layout. Figure 3.2 shows the resulting layout.

Multiple solvers are able to solve layout constraints such as the horizontal box layout. However, different solvers have varying capabilities and perform differently well on this task. As

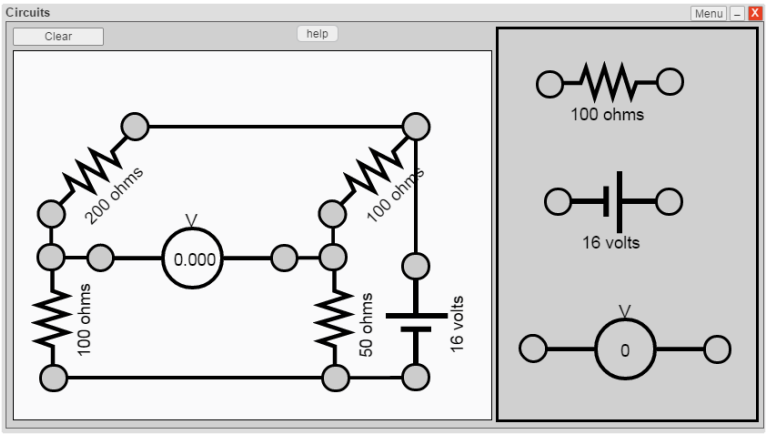


Figure 3.3: Wheatstone bridge simulation in LivelyKernel using Babelsberg/JS

described in Section 2.3, one can use Cassowary as long as the desired constraints are linear. In contrast, Sutherland's relaxation solver can also solve non-linear constraints. In the application the user can choose a solver to solve the layout from a list of available solvers. Using different solvers in such an interactive application allows to understand the difference in their behavior.

Circuits. Physics simulations are yet another typical field of applications of CP. For instance, Figure 3.3 shows an example application of an electrical circuit simulation which contains batteries, resistors, and meters. In this example the user can copy parts by dragging them from the right panel and drop those parts into the circuit. Then, the user can wire parts together through their leads. Each part carries certain physical laws such as Ohm's Law, Kirchhoff's Voltage Law, and so forth. These laws are represented by constraints. As seen in Listing 3.3, constraints are constructed when a part is instantiated. In this example an OO mechanism, inheritance, allows to reuse certain more general constraints. For example, all parts are subject to Kirchhoff's Current Law which states that the algebraic sum of currents meeting at a point is zero. Therefore, the constraint representing this law is defined in the constructor of all part types base class `TwoLeadedObject` in line 5. Then, derived classes can specify part-specific rules in their respective constructor method. For instance, a `Voltmeter` displays the electrical potential difference between its two leads, as specified in line 33. All constraints are linear in this example, so the Cassowary constraint solver is able to solve all constraints occurring in the circuit. Due to its incremental nature, Cassowary can efficiently (re-)solve the constraint system if the user modifies the circuit. This happens every time the user either adds new parts to the circuit or edits values of existing parts.

3.2 Problem Statement

The previous section highlights some examples using Babelsberg/JS. These examples show the power of abstraction that the object constraint programming (OCP) paradigm provides. However, we realize that every example originates from the domain of CP. Accordingly, the examples share some common characteristics. These characteristics make the examples inappropri-

```
1  Object.subclass('TwoLeadedObject', {
2    initialize: function() {
3      this.lead1 = { voltage: 0.0, current: 0.0 };
4      this.lead2 = { voltage: 0.0, current: 0.0 };
5      always: { this.lead1.current + this.lead2.current == 0.0 }
6    }
7  });
8  TwoLeadedObject.subclass('Resistor', {
9    initialize: function($super, resistance) {
10     $super();
11     this.resistance = resistance;
12     always: { this.lead2.voltage - this.lead1.voltage == this.lead2.current * resistance }
13   }
14 });
15 TwoLeadedObject.subclass('Battery', {
16   initialize: function($super, supplyVoltage) {
17     $super();
18     this.supplyVoltage = supplyVoltage;
19     always: { this.lead2.voltage - this.lead1.voltage == supply }
20   }
21 });
22 TwoLeadedObject.subclass('Wire', {
23   initialize: function($super) {
24     $super();
25     always: { this.lead1.voltage == this.lead2.voltage }
26   },
27 });
28 TwoLeadedObject.subclass('Voltmeter', {
29   initialize: function($super) {
30     $super();
31     this.readingVoltage = 0.0;
32     always: { this.lead1.current == 0.0 }
33     always: { this.lead2.voltage - this.lead1.voltage == this.readingVoltage }
34   }
35 });
```

Listing 3.3: Constraints for physical behavior in the circuits demo

ate to evaluate whether Babelsberg achieved its goal of being a useful tool for OO programmers. These common characteristics are discussed in the following.

In the examples, all constraints are typically defined in a single method. Listing 3.1 and Listing 3.2 show this issue for the thermometer and the layout demo. This common pattern assumes that it is possible to define all constraints globally and that the method has access to all relevant components. The method attaches constraints from the outside of objects. It is unclear to which object the respective constraint belongs. While this single method approach is sufficient for small examples, it is not modular and in turn hard to extend.

The circuit example is the only example that supports decentralized constraint construction. By defining constraints in their respective constructors, each class is in charge of defining relevant constraint using their local knowledge. This makes the circuit demo more modular and, thus, more extensible in comparison to the other examples.

For most examples, defining all constraints in a single function is possible, because constraints alone are able to describe the whole application behavior of a typical CP problem. All presented examples are based on some OO code. Then, the respective application introduces several constraints to to restrict those objects. Thereby, the constraint solvers implement all relevant application logic. As a consequence, objects serve as simple data holders. The objects have minor tasks such as render themselves based on the values provided by the

constraint solver. However, this common pattern also assumes that all behavior is expressible as a constraint, and that one or more cooperating solvers are able to solve the resulting constraints. This assumption holds in a typical CP system. In such systems the only thing that matters is state. All behavior to modify the state is encapsulated in the used solvers. In contrast, object-oriented programming (OOP) interweaves state and behavior. Thus, objects become functional units instead of simple data holders.

Because constraints can express the complete logic of a typical CP application, little interaction with the rest of the code is required. This interaction only happens implicitly in the form of state changes. In particular, constraints can only interact with the surrounding system by modifying the system's state, but cannot call dependent components directly. This is no problem, if dependent components continuously read the possibly modified values, e.g. the fabric.js layout example introduces an additional rendering loop. In contrast, Babelsberg integrates poorly with event-based systems, as the rerendering cannot be triggered in this scenario. The main problem here is that Babelsberg modifies primitive attributes of objects directly rather than through the object's application programming interface.

A reason why constraints can solve the whole application logic in the examples is that all examples deal with the same underlying problem, data consistency. Even if the application domains of CP range from layouting to electric circuits, the underlying problem of those domains is data consistency, just in different flavors. In order to let the user interact with the data most examples provide a graphical user interface (GUI) to access and manipulate the underlying model.

Another common observation is that, in nearly every example, constraints are enabled just once and almost never disabled again. In fact, we only found disabling in the form of a complete reset of all active constraints. For instance, when modifying the specified constraints in the layout example, the system resets all previously specified constraints. Even the circuit example which defines constraints in a local scope does not provide a mechanism to unconstrain a single part. Instead, the user can only clear all objects at once. So constraints might be added, but not removed or dynamically changed in the examples, even though such features are available to Babelsberg. The main reason for this is that most examples originate from constraint logic programming (CLP) domain that has no mutable state in nature. In this domain it makes no sense to adapt or disable constraints as this would introduce mutability which does not exist in declarative logic. In contrast, mutability is a familiar concept to most OO programmers. Objects can be consistent or inconsistent, and are able to switch between states. This fundamental difference implies that OO programmers might want to use constraints in a different manner. OO systems are more dynamic that their CP counterpart. Accordingly, constraints in an OO system might demand dynamic adaptation. However, we could not verify the usefulness of dynamic constraints regarding these CP examples only.

Another factor to consider in all previous issues is that most examples are relatively small, so some problems described in this section never occur. For example, a single method that defines all constraints suffices if the system is small enough. However, once the application grows to a certain size, modularity becomes important to manage and maintain the code. Due to their small size, other issues such as maintainability and adaptability are never problematic in the existing examples.

One reason for the small size of the examples is that the purpose of these examples is often to present a particular feature. The use case is often so focused that a single solver is able to solve

the whole problem. For instance, the thermometer example showcases the architecture of cooperating solvers. In order to show this feature clearly, the problem is simple and restrictive. Complex situations in which not everything can be solved with the available constraint solvers never occur, as all examples are from solvable CP domains. In contrast, OO applications incorporate multiple domains and have complex objects and relationships rather than just primitive data types.

To sum up, we discover that all examples available originate from the domain of CP. In consequence, all example problems share some common characteristics such as a global system state and conceptual immutability. These characteristics make it easy to describe the example systems using constraints only. Thus, CP is obviously useful in these examples. Babelsberg can contribute to those problems by providing convenient abstraction using OO mechanisms like encapsulation and information hiding. However, Babelsberg furthermore aims to be a useful tool for imperative programmers. Because CP is fundamentally different from OOP, we cannot conduct whether Babelsberg already fulfills its goal using these examples alone. To verify whether Babelsberg is suited for its goal, we provide a non-trivial application scenario from an OO domain. Using the example which is described in Chapter 4, we want to answer how CP can contribute to OO problems and highlight possible shortcomings of Babelsberg.

4 A Non-Trivial Application Scenario

To evaluate how well the current Babelsberg design fulfills its goal of providing a practical constraint programming (CP) tool for object-oriented (OO) developers, we explore a typical OO application: an interactive online game. First, we describe the desired game mechanics of our game prototype. Then, we shortly discuss possible architectures for the game and in which parts of the implementation Babelsberg comes in handy. Finally, we identify shortcomings in the current Babelsberg design that prevent Babelsberg from being even more useful.

4.1 Game Description

The game which we want to implement using Babelsberg/JS is an adaption of the tanks mini game in Wii Play[1], with some small additions to the game such as the introduction of power ups. Figure 4.1 shows a screenshot of the finished game.

Basic game mechanics. In this game, the player navigates a small toy tank through 2d levels with obstacles. The player tank as well as enemy tanks can fire bullets. The object of a level is to destroy all enemy tanks while avoiding their attacks. When this goal is achieved, the game advances to the next level which has a different combination of enemy tanks and a different level layout. If the player tank is destroyed, the level restarts in its original configuration. Aside from environmental obstacles a level can contain three different types of objects: tanks, bullets, and power ups.

Tanks. Small toy tanks are the main entities in this game. They are either controlled by the player or by the game in case of enemy tanks. When a level starts, tanks are placed according to the current level into the environment. Tanks have a direction of movement and move into that direction with a constant speed which can also be zero. A tank can rotate its body and turret independently. This makes it possible to move and aim at the same time. Tanks can shoot a number of bullets at the same time. After its launch a bullet moves independently from its respective tank. If the number of bullets launched by a tank exceeds that tank's maximum number of bullets, this tank is unable to shoot until one of its bullets is destroyed. Additionally, it is not possible shoot directly into a wall.

Bullets. Bullets are launched by tanks and then keep flying into the direction they were launched. When a bullet hits a wall, it may ricochet off that wall. However, it may only ricochet a specific number of times, once by default. The ricochet is subject to the law of reflection which states that the direction of an incoming object and the direction of the reflected object

[1] Nintendo Co. Ltd., Wii Play, `http://www.nintendo.com/sites/software_wiiplay.jsp` (last accessed April 30, 2015)

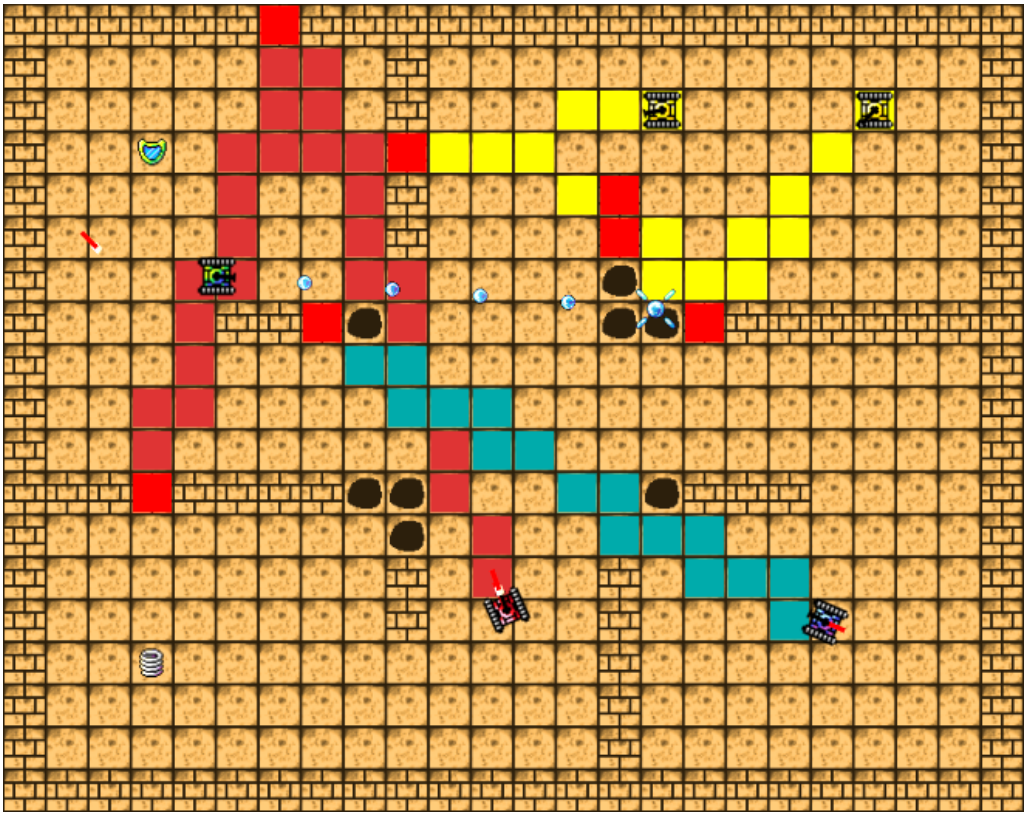


Figure 4.1: Screenshot of the sample application

have the same angle with respect to the surface normal. This reflection is the only way to adjust a bullet's direction. Bullets destroy tanks or other bullets they touch. When that happens, the bullet itself is also removed from the environment. Bullets can also destroy the tank that launched them. The speed of bullets varies and is determined by the type of the shooting tank. However, fast bullet, despite their speed, share the same properties as other bullets. For example, they can be destroyed by any other bullet.

Power ups. Power ups are objects that temporarily give tanks particular abilities. Power ups appear at the start of a level in a level-specific pattern. Each tank can collect a power up by touching it. Then, the power up entity disappears and benefits the tank for a certain amount of time. Bullets fly through power up without interacting with them. Power ups are static entities, i.e. they do not move on their own. Tanks can have multiple power ups of different types at the same time, combining their benefits. If a tank collects an already active power up, the effect duration is prolonged. The game features three types of power ups: a shield, a spring, and a slime power up.

- The shield makes the tank invincible. As long as this power up is active, bullets cannot destroy the tank.
- The spring provides the tank one additional bullet ricochet, i.e. all bullets the tank launches bounce off walls one additional time while this power up is active.
- The slime affects all other tanks. They cannot move while this power up is active.

Table 4.1: Tank type statistics in our example game

Tank	Speed	Bullets	Ricochets	Bullet Speed	Movement Intelligence	Aiming Intelligence
Player (Green)	Normal	3	1	Normal	Varies	Varies
Yellow	Immobile	1	1	Normal	-	Random
Red	Slow	2	1	Normal	Avoids walls	Mildly seeks player
Blue	Slow	1	0	High	Avoids walls & bullets	Strongly seeks player

Interactions. The previously described entities interact in several different ways depending on their types. The following enumeration describes the concrete interactions:

- Tanks push each other if they collide.
- If a tank touches a bullet both are destroyed and removed from the environment.
- Tanks collect power ups by touching them. The power up disappears and gives the respective tank a power up-dependent effect for a certain amount of time.
- As with tanks, bullets destroy each other once they collide.
- Bullets and power ups as well as power ups themselves do not interact at all.

Levels. The levels in which the game takes place are tile-based fields. Each level is composed of three types of tiles: walls, holes and plains. These tiles differ in what type of entities can pass it and which entities collide with tiles, as described in the following:

- Walls are impassable by any entity.
- Holes are impassable for tanks, but bullets can fly over holes.
- Plains are passable by any entity.

Controls. The player controls the movement of its tank using the arrow keys. The movement is limited to 8 directions. Aiming and shooting are handled via mouse input. More precisely, the player tank's turret always points into the direction of the mouse pointer. Additionally, a crosshair graphic follows the mouse and indicates the direction in which the tank would shoot bullets. Clicking the left mouse button causes the tank to launch a bullet into the direction of the mouse pointer.

Enemy types and behavior. The game features four different kinds of tanks which can be identified by their color. The abilities and the behavior of the different tank types vary, as stated in Table 4.1. For example, the blue enemy tank moves at slow speed but shoots fast bullets. In its movement the blue tank avoids walls as well as bullets if necessary. Its turret always faces towards the player tank, because with zero bullet ricochets the blue tank can only hit the player directly. The number and combination of enemy tanks vary depending on the current level.

Modes. At any given point in time, the player has the possibility to switch to the editor mode. In the editor mode the game is paused, yet, the current level is still visible. In addition to this, a tile-based cursor appears. By clicking the left mouse button the player can modify the type of the tile under the cursor. Using this feature, the player can adapt the shape of the level as seen in Figure 4.2. Additionally, our game features a debug mode. In debug mode the game renders additional information such as the velocities of the entities and their line of fire. Both

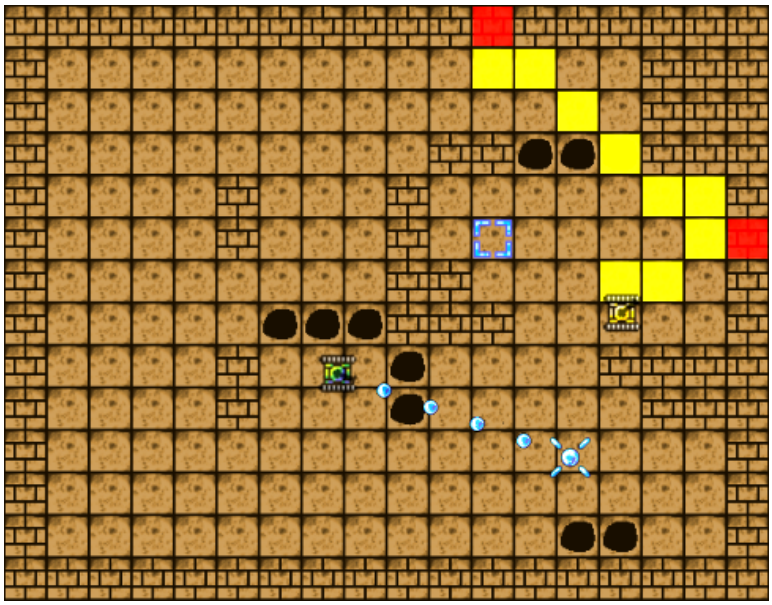


Figure 4.2: Screenshot of the editor mode

modes, the debug mode and the editor mode, can be activated and deactivated by the press of a button.

4.2 Design

This section briefly summarizes the most important parts of this game's implementation. We have chosen a direct approach for this game's implementation in keeping with common JavaScript practices. An alternative, more elaborate architecture is given in Section 4.3. When the game starts, all level descriptions and assets are loaded. When the loading finishes, we initialize a `game` object and start a simple game loop. Upon initialization, the `game` instructs the creation of relevant services such as an `Input` service as well as a `Renderer`. The `game` keeps track of the current level. A `World` instance represents the level which is constructed by a dedicated builder. Additionally, the `game` is in charge to delegate the game loop to relevant components and services.

The `World` represents the current level and keeps track of a tile-based `Map` as well as all `GameObjects` as illustrated in Figure 4.3. Each `Tile` represents a specific terrain that is passable by certain types of `GameObjects`. We provide physics and rendering functionalities in `GameObject`, the superclass for all entities, because all entities described in Section 4.1 share those functionalities. We use simple sprite sheet-based animations for the rendering of `GameObjects`. Each graphical asset is divided into multiple frames which a `Renderer` draws over time according to the `frameSequence` of the respective `Animation`, as depicted by Figure 4.4. For the physical behavior the `GameObject` has corresponding attributes such as a `position`, a `direction` and a `speed` attribute as well as a `radius` for collision detection. During each frame the `position` is updated according to the object's `direction` and `speed`. Manipulating the physical behavior is the task of each respective subclass. Additionally, `Tanks` contain attributes to handle its turret, we also make use of a separate `Controls` object. This object handles the behavior of the `Tank`

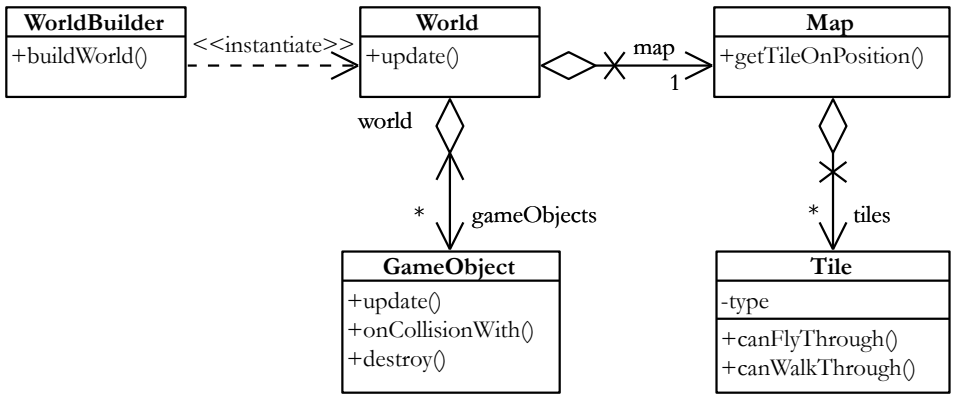


Figure 4.3: Composition of a level

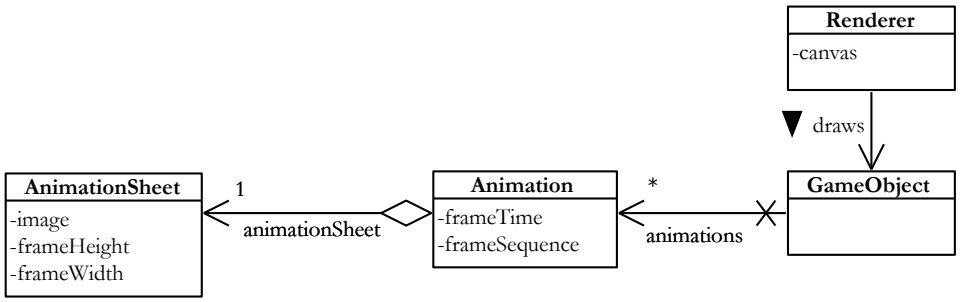


Figure 4.4: Classes responsible for the rendering process

which is either controlled by the player or the program. Furthermore, `Tanks` hold all necessary information to be able to instantiate `Bullets`.

4.3 Design Alternative

One possible alternative design is to use an architectural pattern like an entity component system which was introduced by the game development community[2]. An entity component system is a data-driven approach that consists of three types of objects to handle the game logic: entities, components, and systems. An entity refers to a coarse game object as a separate item and serves as a bag of components. These components are simple data classes for a specific aspect of that entity. Attaching a component to an entity labels it as possessing this particular aspect. Systems are the functional units of this design. Each system refers to the subset of all entities that posses certain aspects, i.e. components. For example, suppose a `TimeIntegrationSystem` refers to all entities that hold a `Position` and a `Velocity` component. Then, in

[2] Scott Bilas, A Data-Driven Game Object System, Game Developer Conference 2002, http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides.pdf (last accessed April 30, 2015)

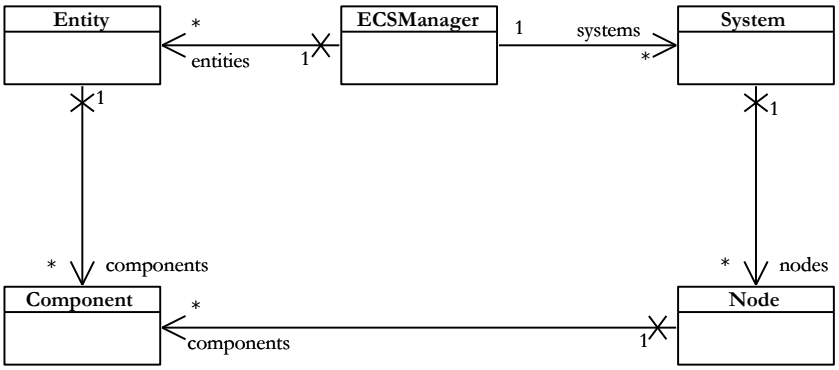


Figure 4.5: Main components of an entity component system

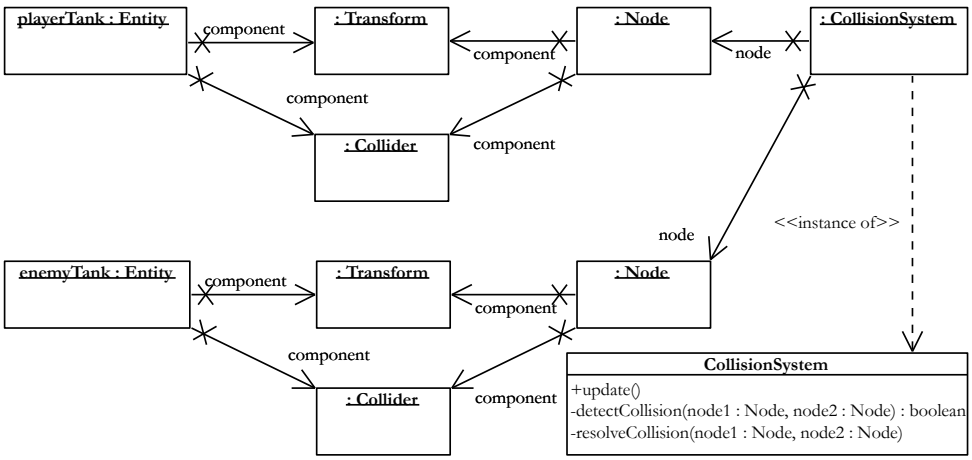
each time frame the system iterates through all matching entities and adds the velocity to the position vector. Many entity component systems like the Ash framework[3] introduce an additional type of objects, called nodes. Nodes ensure that systems only access the components they are allowed to as nodes only reference the components specified by the system. This leads to a coarse-grained architecture as illustrated in Figure 4.5.

The concept of entity component systems overcomes the two major shortcomings of the entity component pattern[4]. The first major shortcoming is the feature envy of components on some data handled by another component. A common example is the position attribute that is typically part of a physics-related component. A rendering component needs access to this attribute in order to render the entity appropriately. However, the rendering component can only access the respective data through the physics component. This violates the concept of independent components. A second shortcoming is that the entity component pattern cannot handle interactions between multiple entities well, as this pattern focuses on independent components. In contrast, an entity component system can handle this issue easier because systems can access all relevant entities.

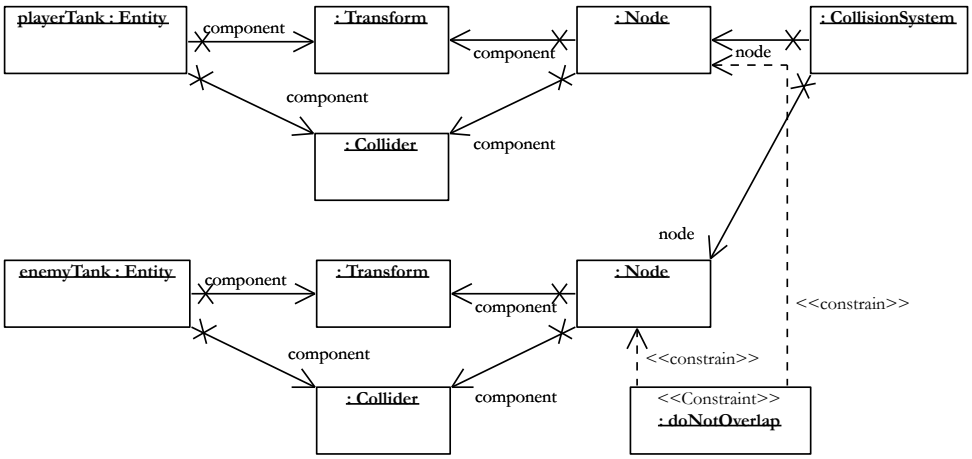
Entity component systems offer an architecture suited for game programming. However, they are inappropriate to verify our initial question, whether Babelsberg is suited for OO programmers. This is due to the fact that entity component systems are data-driven. Therefore, entity component systems share several characteristics with traditional CP systems as discussed in Section 3.2. Suppose our game example to involve a `CollisionSystem` as depicted in Figure 4.6. To handle collisions the `CollisionSystem` refers to entities that have a `Transform` as well as a `Collider` component attached to them. In this case, the `playerTank` and the `enemyTank` `Entity` both match that condition. In its `update` routine the `CollisionSystem` iterates all matching entities through the according `Node` objects. The `CollisionSystem` calls its `detectCollision` method to check whether a collision exists. If that is the case, a second method `resolveCollision` handles the collision appropriately by modifying the data stored in the components. As this example illustrates, in an entity component system architecture systems encapsulate all functionality, just like constraint solvers do in CP. As a consequence, components are simple

[3] Richard Lord, Ash Entity System Framework, http://www.ashframework.org/ (last accessed April 30, 2015)

[4] Bob Nystrom, Game Programming Pattern: Component Pattern, http://gameprogrammingpatterns.com/component.html (last accessed April 30, 2015)



(a) Collision handling in an ordinary entity component system



(b) Collision handling in an entity component system using constraints

Figure 4.6: Two variants of collision handling in an entity component system

data holders which do not implement any game logic. With this intentional separation of data and behavior entity component systems become too similar to Babelsberg's previous examples described in Section 3.1.

A second reason for entity component systems to be inappropriate for us is that constraints introduce logic to nodes and components. Suppose we want to express the functionality of the `CollisionSystem` using constraints. To do so, we can specify a constraint that the distance between the positions of the `playerTank` and the `enemyTank` should be greater than the sum of their radii, assuming that their `Colliders` are spheres. Now, the logic of the `detectCollision` and `resolveCollision` methods is moved to a constraint attached to nodes and components, as illustrated in Figure 4.6. However, this is contrary to the idea of entity component systems because the constraint introduces game logic to the data objects. Additionally, it is unclear which object initialized the constraint and at what point of execution. So, instead of using a dedicated architecture we prefer a simpler and more object-oriented design.

4.4 Involving Constraints

This section highlights possible parts of the application that can benefit from the usage of constraints. As described in Section 2.3, most solvers are limited to a specific domain. So, these solvers only apply to specific types of problems. While building our game, we found its physics an appropriate problem to be expressed as constraints. Listing 4.1 shows a constraint that takes care of the collision detection and resolution between tanks. Line 2 states that the distance between the two tanks should be greater than the sum of their radii, i.e. they do not overlap. Sutherland's relaxation solver is appropriate to solve the constraint. However, we discover two shortcoming when using the constraint. First, the constraint would only cover a collision between two `Tank` objects. In contrast, the collision between any other two types of `GameObjects` involve custom behavior and, therefore, cannot be solved by any available solver. Nevertheless, the collision detection mechanism for all cases involve the same condition. Letting Babelsberg handle the collision between two `Tanks` while manually detecting and resolving all other collisions seems inconvenient with regards to code reuse. Second, using relaxation the collision resolution results in imprecise or unstable collisions. In contrast, we expect a precise pushing behavior. Our solution to overcome both issues is described in Section 7.1

```
1 always: { solver: new Relax()
2   tank1.position.distance(tank2.position) >= tank1.radius + tank2.radius
3 }
```

Listing 4.1: Specify that two tanks should not overlap

As stated in Section 3.2, constraints are useful to ensure data consistency. In OO programs these data consistency needs are typically solved by a data flow mechanism. This fact suggests that we can use constraints to declaratively model data flow in our application. Due to its ability to support arbitrary domains, DeltaBlue seems to be the best fit for this task. To illustrate this, consider the following situation in our example: In the game we make use of two different coordinate systems, a screen coordinate system and a world coordinate system. The input as well as the rendering are handled in screen coordinates, the game logic takes place in world coordinates. A camera-like `Viewport` manages the segment of the `World` that is visible for the

user. Therefore, the `Viewport` is responsible for converting between screen and world coordinates. To determine which segment of the `World` is rendered, the `Viewport` has two attributes, the `middlepoint` and the `extent` of the segment. For convenience, we want to access the position of the mouse in world coordinates without explicitly converting the mouse position to world coordinates on every access. To do so, we can define the constraint stated in Listing 4.2. Line 2 specifies that the position of the input should be always equal to the mouse coordinates converted to world coordinates. As stated in Section 2.3, Babelsberg can automatically derive a one-way propagation function from the given constraint to solve the constraint using DeltaBlue. So, DeltaBlue enables us to specify unidirectional data flows using high-level abstraction, i.e. constraints state what the data flow should achieve.

```
1 always: { solver: new DBPlanner()
2   input.position.equals(viewport.screenToWorldCoordinates(input.mouse))
3 }
```

Listing 4.2: Constraint to ensure that the mouse position on screen and the respective position in world coordinates are consistent

In contrast, using ordinary data flow mechanisms one has to explicitly specify the data flow with its starting point, its end point, and the mapping function. In Listing 4.3 we use `AttributeConnections` available in LivelyKernel [29] to create an equivalent data flow. One can create an `AttributeConnection` using the method `lively.connect`. This method gets an object and a field name as the first two parameters to define the starting point of the data flow. When this value is changed, the fifth parameter, a mapping function, is applied to the new value. Then, the result is assigned to the object's attribute specified by the third and fourth parameter. Note, that we actually have to define three `AttributeConnections` to correctly update the world position of the mouse. Lines 1 to 3 create an `AttributeConnection` to listen and update on a mouse movement. However, the `Viewport` itself can also be modified. So, we need the two additional connections from line 4 to 9 to cover this case. This explicit data flow can potentially violate object encapsulation and information hiding as it requires the programmer to know the internal structure of objects. Another fact to consider is that the mouse position as well as the `Viewport's` attributes are complex objects, vectors. So in addition to the existing `AttributeConnections`, we have to explicitly install and maintain connections for each specific coordinate of the existing vectors. Babelsberg takes care of this issue by default, because listeners are added to all relevant attributes. Additionally, Babelsberg maintains the listeners automatically if a complex object is assigned.

```
1 lively.connect(input, 'mouse', input, 'position', function(screenPosition) {
2   return viewport.screenToWorldCoordinates(screenPosition);
3 });
4 lively.connect(viewport, 'middlepoint', input, 'position', function() {
5   return viewport.screenToWorldCoordinates(input.mouse);
6 });
7 lively.connect(viewport, 'extent', input, 'position', function() {
8   return viewport.screenToWorldCoordinates(input.mouse);
9 });
```

Listing 4.3: Data flow to ensure that the mouse position on screen and the respective position in world coordinates are consistent

We discover that a programmer is more encouraged to store redundant data when using constraints as in the previous example. Typically, programmers avoid redundant data as one has to keep the data consistent manually. However, Babelsberg allows to model the relationship between data explicitly. Therefore, data consistency is achieved more easily and redundant data become less error-prone. In addition to the presented example, we make use of DeltaBlue to ensure data consistency in the following parts in our game:

- The crosshair graphic should always stay at the mouse screen position.
- Additional target line graphics are placed between the crosshair graphic and the player tank interpolated by specific values.
- The player tank's turret should always point to the mouse position.
- In editor mode the tile-based cursor should always point to the tile under the mouse pointer.

4.5 Open Issues in Babelsberg Design

In the previous section we highlight how Babelsberg can be applied in a classical OO environment. Yet, we also discover a couple of shortcomings that prevent Babelsberg from being even more useful. In this section we discuss concepts missing in Babelsberg to fit the requirements of a typical OO environment.

Constraints without capable solvers. Constraints provide the largest benefit when addressing properties that are easy to describe, yet hard to solve. To solve and maintain constraints CP relies on the power of the available constraint solvers. However, most solvers are limited to a set of primitive types. In contrast, many problems in an OO environment deal with complex objects. Suppose we want to specify a constraint to avoid a tank from driving through walls and holes. The respective constraint is described in Listing 4.4. The constraint is easy to understand, but the method `getTile` involves an array lookup for a `Tile` depending on the `position` of the tank. To solve this constraint, the tank's `position` has to be updated in a way that the array lookup would return an appropriate `Tile`. Unfortunately, no available solver is capable of such an implication. Such specific problems often occur while programming in an OO environment. As a consequence, a practical way to deal with constraints that no available solver can solve is needed.

```
1 always: { map.getTile(tank.position).canWalkThrough() }
```

Listing 4.4: Hypothetical constraint to ensure that a tank moves only in passable tiles

Strict separation of state and behavior. In CP only state matters as described in Section 3.2. All necessary behavior to achieve the desired state is encapsulated inside the respective solvers. As a consequence, CP strictly separates state and behavior. In addition, the user of a CP language can only specify the desired state, but no functionality. This approach is fundamentally different from object-oriented programming (OOP) which connects state and behavior. Suppose we want to define the following functionality: when a bullet hits a wall, then the bullet bounces off this wall. In ordinary CP systems this functionality is not expressible as it would

involve user-defined behavior, yet, most solvers only have limited and predefined logic. Defining this behavior in OO language requires implicit checks and calls to the desired functionality. This is similar to the fact that OOP maintains desired constraints implicitly through scattered code fragments. The object constraint programming (OCP) paradigm addresses this particular issue by making the constraints explicit. Yet, OCP does not allow statements about the desired behavior. To better integrate constraints and OOP a way to connect desired state and behavior is desirable.

A convenient scoping mechanism. As stated in Section 3.2, earlier examples using Babelsberg originate from the domain of constraint logic programming (CLP). In the domain of CLP adapting or disabling constraints would introduce mutability. However, mutability is a concept that does not exist in declarative logic. In contrast, mutability is a familiar concept to most OO programmers. OO systems change during the execution of a program and can become consistent or inconsistent. For example, certain properties may only be desired for a portion of the program execution. Babelsberg already supports the activation and deactivation of a constraint. However, constraints are currently manipulated using the low-level methods `enable` and `disable` on a constraint object. Using these methods, one has to create a constraint and, at some later point in the execution, explicitly activate the constraint depending on whether an if-condition evaluates to true. To safely disable constraints one has to wrap the respective portion of the code in a try-finally block. As one want to frequently enable and disable constraints, such an explicit manipulation of constraint objects quickly becomes unfeasible. So, a structured and high-level construct to enable constraints only during a particular scope is desirable. In the spirit of Kaleidoscope Babelsberg/R supports an `assert-during` construct which takes a closure and activates a constraint during the evaluation of the closure. While this mechanism allows for control-flow specific constraints, one would also want to manipulate constraints based on the current context of the system, e.g. disable some constraints while a level is edited. Surely, constraints should allow or even encourage dynamic activation and deactivation. Therefore, a practical way to scope constraints is required.

5 Refining Babelsberg Concepts for Object-Oriented Environments

Section 4.5 identified three shortcomings in the current Babelsberg design: a missing mechanism to deal with constraints unsolvable by any available solver, the missing ability to invoke or adapt behavior based on constraint, and the need for a convenient scoping mechanism. This chapter describes concepts to address these shortcomings.

5.1 Continuous Assertions

One of the fundamental assumptions of constraint programming (CP) is that problems are typically easy to describe, but hard to solve. To solve and maintain desired properties CP makes use of constraint solvers. However, these solvers have limited domains and functionality, even when combining multiple solvers using an architecture of cooperating solvers as in Babelsberg. So, some problems cannot be solved using the available constraint solvers, especially when dealing with high-level objects in object constraint programming (OCP).

When a constraint is not solvable by any available constraint solver, the programmer has two possibilities. First, the programmer could solve and maintain the invariant manually as in an ordinary object-oriented (OO) language. However, doing so one would lose all advantages of OCP even with OCP available. Second, the programmer can encapsulate the needed functionality in a new solver. In addition to problem-specific solving routines, the programmer has to implement solver specific constraint variables and primitive constraint objects required for the constraint construction. So, writing a specific solver for each problem puts an additional load on the programmer and, therefore, seems unfeasible for typical OO programmers.

To find a practical solution to this problem, we investigate how the `always` statement works. As depicted in Figure 5.1, defining a constraint with predicate `p` separates the domain of the system state into two categories, valid and invalid system states. The figure shows the successive states in the execution of a program. An OCP application is typically in a valid state, as defined by its constraints. When an assignment moves the system state into the invalid portion, a constraint solver is called in order to (re-)solve the violated constraint. So, the system moves back into a valid state in terms of constraints. Note, that this program state is typically different from the previous one, because assignments are modeled using temporary constraints. The invalid state is temporary and only visible by the constraint system.

At its core the `always` statement prevents the constraint expression from becoming invalidated by employing constraint satisfaction techniques. If no solver is available to maintain a certain constraint, another solution is required to bring the system back into a consistent state. One solution to this issue is to revert the system back into its previous state. So, if an assignment attempts to invalidate a certain constraint, the assignment will be discarded. If the condition is already violated on constraint construction, the construction of the constraint will fail.

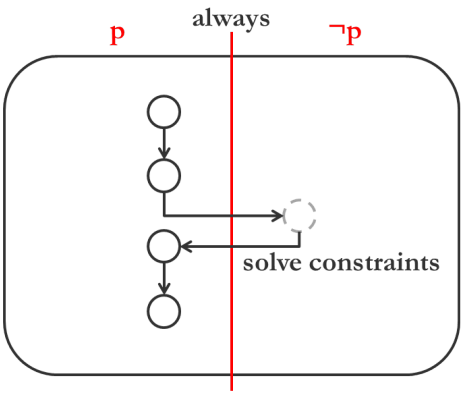


Figure 5.1: System state trajectory adjusted by a constraint created using the always statement

Using this concept allows us to describe expected invariants of our system. Then, Babelsberg checks the expected invariants during the execution of the program. This behavior is similar to the assertion statement which is available in many programming languages. Similar to assertions we check for a specific condition. In spite of this, assertions only check a condition at a single point during execution, whereas our statement checks the desired invariant continuously. We call this concept *continuous assertion*.

We argue that for certain types of problems, preventing the system from entering an erroneous state suffices. In contrast to the `always` statement which enforce invariants, continuous assertions check for and prevent invalid system states without any repair attempts. In spite of this, both statements are quite alike. This fact suggests a similar notation. This unified notation is further covered by Section 5.4.

One should only use continuous assertions if the following two criteria are met.

- No available solver is able to deal with the constraint in a sufficient way.
- Reverting invalid statements is a valid option for the program. Continuous assertions can revert assignments, yet, this behavior might be unfamiliar to OO developers.

5.2 Reactive Constraints

As stated in Section 4.5, CP and object-oriented programming (OOP) treat behavior differently. While CP strictly separates state and behavior, OOP integrates them. As OCP attempts to unify both paradigms, it has to deal with this fundamental difference. Babelsberg already allows to use arbitrary function expressions to define constraints, yet, constraints can only modify the state of a program, not its behavior. In this section we describe concepts to connect state described by constraints and OO behavior to push the integration of OCP forward. From our example, we identified two distinct cases how behavior can be related to constraint expressions: invoke a piece of behavior *as soon as* a constraint expression evaluates to true, and adapt behavior *as long as* an associated constraint expression holds. In the following we discuss both cases separately.

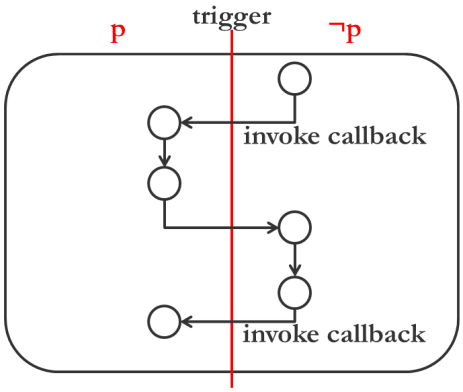


Figure 5.2: A trigger constraint invokes the given callback when the system reaches a state in which the corresponding constraint expression evaluates to true

5.2.1 Trigger Constraints: Invoke Behavior Depending on Constraint Expressions

Similar to the maintenance of invariants, the invocation of behavior upon a condition is typically managed implicitly by scattered code fragments in OOP. For example, a certain event should be emitted once a condition is met. OCP treats the first issue by making the desired constraint explicit. We address the second issue, triggering behavior on a condition, in a similar way, i.e. by making the desired functionality explicit. Therefore, we introduce the concept of *trigger constraints*. Using a trigger constraint, one can explicitly specify that a given piece of behavior should be invoked once a given condition evaluates to true.

```
1 predicate(function() {
2   return input.pressed("leftclick");
3 }).trigger(player.fireBullet.bind(player));
```

Listing 5.1: Trigger constraint to launch a bullet once the player presses the left mouse button

As an example, suppose we want to express the following functionality of our game: when the player presses the left mouse button, then the player tank should launch a bullet. Listing 5.1 shows how a trigger constraint specifies this behavior using the unified notation described in Section 5.4. Line 3 specifies the callback that should be invoked. In this case, the `fireBullet` method should be called with the player bound as `this` context. Line 2 shows the constraint expression of the trigger constraint, namely, whether the input service recognizes a left click of the player. Figure 5.2 depicts when the callback is called depending on the result of the constraint expression. As shown in the figure, every time the system enters a state in which the constraint expression is fulfilled from a state in which the constraint expression is not fulfilled, Babelsberg invokes the callback. So, before the callback can be called again, the system has to leave the space of states in which the constraint expression is fulfilled. When reentering this space, the given callback is called again. When this happens, Babelsberg invokes the callback before execution of the next statement. If the constraint expression is already fulfilled during the construction of the trigger constraint, the given callback is immediately invoked.

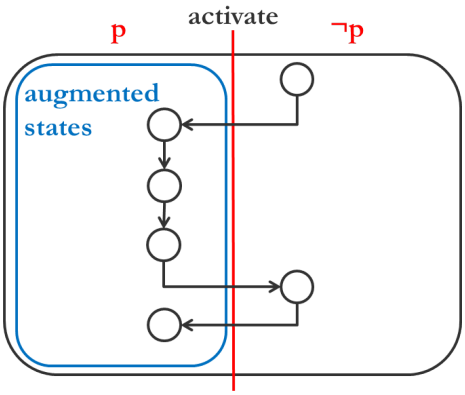


Figure 5.3: System state trajectory augmented by an activator constraint

5.2.2 Activator Constraints: Adapt Behavior Depending on Constraint Expressions

Games frequently involve dynamic changes of the system's behavior depending on various conditions. Certain functionality can be restricted to a specific game mode, e.g. velocities are only rendered in debug mode. As with trigger constraints, we want to specify the relation between the behavior adaption and the condition explicitly. So, we introduce the concept of *activator constraints*. Using activator constraints, one can specify that a certain adaption in behavior applies as long as a given condition holds.

```
1 var powerUpLayer = new Layer().refineObject(tank, {
2   getBulletRicochets: function() {
3     return cop.proceed() + 1;
4   }
5 });
```

Listing 5.2: Constructing a COP layer with partial behavior

Note, that activator constraints rely on an additional concept to dynamically adapt behavior. The concept of context-oriented programming (COP) [7, 21] provides such a mechanism. COP provides means to associate partial behavior definitions with explicit context objects, *layers*. Layers can be activated or deactivated during the execution of a program, usually based on the control flow. When a layer is activated, the partial behavior definitions become part of the respective objects and classes until the deactivation of the layer. COP enables dynamic adaption and variation in behavior based on the current context. To illustrate this, consider the spring power up described in Section 4.1. The spring power up causes all bullets of the affected tank to bounce off walls one additional time. This functionality can easily be expressed using layers as in Listing 5.2. Line 1 creates a layer and associates a partial object definition to this layer. Lines 2 to 4 augment the method `getBulletRicochets` of the `tank` object by incrementing the result of the original function by one. In addition to the definition of the layer and the partial behavior, one has to activate the layer in order for the adaption to take place. A layer is typically de-/activated globally or for the dynamic extent of a function invocation. Such mechanisms only allow control flow-specific scoping. However, adaptions can also depend on scopes other than control flow such as conditions on the current state of the program. As with trigger constraints, we want to treat the condition associated with a context explicitly.

We imagine activator constraints to activate a given layer as long as the associated constraint expression is fulfilled, as illustrated by Figure 5.3. Once we reach a state in which the constraint expression is fulfilled, the layer is activated until the constraint expression evaluates to false. Then we deactivate the respective layer again.

As a continuation of the spring power up example, suppose the benefit of the power up is of limited duration. So, the power up is only active until its time is running out. We use a dedicated `Timer` class to keep track of time in our example. Using activator constraints, we can connect an instance of this class with the previously defined `powerUpLayer`, as shown in Listing 5.3. In line 2 we define the condition that the `remainingTime` attribute of the `timer` is greater than zero. Next, we connect this condition with the `powerUpLayer` and, thereby, create the desired relation in line 3.

```
1 predicate(function() {
2   return timer.remainingTime > 0;
3 }).activate(powerUpLayer);
```

Listing 5.3: Activator constraint to activate a power up for a specific amount of time

We use layers as units of dynamic behavior adaption because of the following reasons:

- COP treats context objects, layers, explicitly. These context objects can easily be activated based on a condition.
- COP is no special solution dedicated to JavaScript, but has been implemented in many other languages such as Lisp, Smalltalk, Python, Java, and Ruby. So, Babelsberg implementations in languages other than JavaScript can also base on the very concept of COP.
- Aside partial object and class definitions COP layer can act as units of scoping for constraints as well, as described in Section 5.3. Being able to use constraints to activate layers and layers to scope constraints provides additional synergy effects between both concepts.

Appeltauer et al. compare several COP implementations as well as their activation means [1]. We argue that constraint expressions represent a flexible layer activation mechanism which is decoupled from the control flow of the program. Section 8.2 deals with the comparison between activator constraints and event-based as well as implicit layer activation techniques.

5.3 Scoped Constraints

As stated in Section 3.2, CP typically focuses on describing a global system state. In contrast to CP, OOP revolves modularization of system state and dynamic adaption. Babelsberg already provides means to dynamically adapt constraints. Still, these means are limited to the low-level methods `enable` and `disable`. As explained in Section 3.2, manipulating constraints using such low-level techniques is unfeasible for larger systems. In case of Babelsberg/R, one can activate constraints during the dynamic extent of a code block. Yet, this mechanism is control flow-specific. In Section 5.2.2 we introduced activator constraints as a mechanism to activate COP layers based on arbitrary conditions. So far, layers have been used as convenient units of scoping to adapt behavior. However, as we integrate constraints and OO concepts, the requirements of an OO environment start to apply to constraints as well. Constraints should be able to adapt dynamically as behavior does. So, similar to partial class and object behavior, we in-

troduce *scoped constraints*, i.e. constraints associated with a COP layer. One can associate layers with constraints as one would do with partial behavior. Similar to partial behavior definitions, scoped constraints take effect as soon as the associated layer becomes active. Deactivating the layer again causes the constraints to be disabled. That way, the constraint is enabled as long as the associated layer is active.

To exemplify the concept of scoped constraints, suppose the following functionality of our game example for the tile-based cursor in the editor mode: as long as the game is in editor mode, this cursor should always refer to the `Tile` under the mouse pointer. To cleanly separate the editor mode from the rest of the game, the editor mode is represented by a dedicated layer, the `EditorLayer`. This layer crosscuts all relevant modules to implement the editor mode, including the constraint described above. Listing 5.4 shows the definition of this constraint. First, line 1 states that the following predicate is associated with the `EditorLayer`. Line 2 specifies that the `tileIndex` of the editor mode `cursor`, and so its position, should be equal to the mouse position converted to tile-based coordinates. Finally, lines 3 to 5 ensure that this constraint expression will be fulfilled using an instance of the DeltaBlue constraint solver. The whole definition causes the desired behavior to only take place in editor mode.

```
1 EditorLayer.predicate(function() {
2   return cursor.tileIndex.equals(map.positionToCoordinates(input.position));
3 }).always({
4   solver: new DBPlanner()
5 });
```

Listing 5.4: Scoped constraint to make the cursor graphic hover over the tile under the mouse pointer while in editor mode

As an additional advantage, layers can group multiple constraints due to their cross-cutting aspect. So, one can structurally en-/disable multiple constraints related to the same context. In conclusion, COP provides constraints the variability needed in an OO environment.

5.4 A Unified Notation for Constraint Specification

Currently, most Babelsberg implementations provide two new primitives for constraint construction, `always` and `once`. For instance, the `always` expression followed by a block with the constraint expression is used to construct a global constraint in JavaScript. To do so, an appropriate call to the global `always` function is created via a source code transformation. All constraints constructed through `always` are enabled immediately. However, this type of definition only allows to define global constraints as usual in CP applications. As we introduce trigger, activator, and scoped constraints to better fit constraints into OO systems, a unified notation to cover all presented concept is desirable.

In contrast to the currently used notation for constraints, we treat constraint expressions explicitly. To do so, we propose a single global method, `predicate`. This method takes a function as parameter and returns a corresponding `ConstraintExpression` object. A `ConstraintExpression` provides methods to construct actual constraints:

- The `once` and `always` method return an ordinary constraint. Options, e.g. the solver to solve the constraint, are passed as parameter as in the old notation.
- The `assert` method returns a continuous assertion instead of an ordinary constraint.

- The `trigger` method takes a callback as parameter and returns the respective trigger constraint. Listing 5.1 shows an example of this method.
- The `activate` method takes a layer as parameter and returns the respective activator constraint. Listing 5.3 exemplifies the usage of this method.

To enable scoping, we extend layers to provide an additional method, `predicate`, that works like its global counterpart but returns a scoped constraint expression instead of a global one.

We argue that the described notation using constraint expressions as separate entities has several advantages:

- The notation allows to reuse a constraint expression. With the introduction of trigger and activator constraints reusing the very same constraint expression becomes plausible. For instance, suppose an activator to activate a layer. If the system needs additional setup for the behavior variation, this setup could be provided using a trigger constraint on the same constraint expression.
- The notation allows to decouple the constraint declaration from the actual usage. In contrast, the old notation tightly bound both together. So, constraint expressions become a first class entity. Functions can take constraint expressions as parameters or return them. For example, Listing 5.5 shows how the predicate used in Listing 5.3 can be provided by a dedicated function. This allows to hide the concrete implementation of the constraint expression inside the object that provides the constraint expression.

```
1  Object.extend(Timer.prototype, {
2    untilTimeout: function() {
3      this._untilTimeoutExpression = this._untilTimeoutExpression || predicate(function() {
4        return this.remainingTime > 0;
5      });
6
7      return this._untilTimeoutExpression;
8    }
9  });
```

Listing 5.5: Expression that a `Timer` has not reached a timeout encapsulated in a dedicated method

5.5 Open Issues

Integrating the concepts of COP and CP leads to several issues about their semantics. Each question allows for multiple valid answers as the expectation of the programmer depends on the concrete use case. None of the following three issues is relevant to our game use case.

The first issue is concerned with what happens when two activator constraints refer to the same layer. This question can be solved in multiple ways:

1) The layer is active if one or more of the activator constraints evaluate to true.
2) The layer is only active if all associated activator constraints evaluate to true.
3) Treat an activator constraint as an equivalent always constraint as in Listing 5.6. Line 2 ensures that Babelsberg keeps the layer active if the constraint expression becomes true. When a second activator constraint is defined, the values of `layer.isActive()` and both activator constraints are maintained to be equal by Babelsberg.

4) Trying to define an activator constraint for a layer that is already associated with an activator constraint raises an error.

```
1  predicate(function() {
2    return layer.isActive() == activatorConstraint.isFulfilled();
3  }).always();
```

Listing 5.6: An activator constraint modelled as an always constraint

The second issue regards the integration of activator constraints and control flow-based activations of a layer. As a concrete question: what happens if a layer that is associated with an activator constraint is activated manually?

1) Trying to manually activate a layer that is associated with an activator constraint raises an error.
2) The manual layer activation fails silently.
3) The activator constraint is modelled as an always constraint. The manual activation causes Babelsberg to solve the constraint expression of the activator constraint.

Note, that the semantics of both issues are clear when using the layer activation semantics of ContextJ [2] or JCop [3]. In these implementations of COP activating a layer multiple times causes the variation in behavior to be applied multiple times. So, activating a layer twice, regardless whether the activation is manual or by the means of activator constraints, applies the behavior adaption twice.

For the third issue, suppose an activator constraint refers to a layer. The constraint expression of the activator constraint evaluates to true and, therefore, the layer is active. What should happen when the activator constraint is now disabled?

1) Disabling an activator constraint deactivates the layer.
2) Disabling the activator constraint has no effect on the layer. The layer holds its current state and stays active even if the constraint expression of the activator constraint becomes false, because the activator is disabled.

Currently, our implementation holds the last state of the layer until the layer is adjusted manually or by the means of another activator constraint as in the first option.

6 Implementation

This chapter explains the implementation of the concepts described in Chapter 5 in JavaScript. The implementation of these concepts requires the extension of two existing libraries: Babelsberg/JS [12] and ContextJS [28]. First, we explain the construction and maintenance of constraints in Babelsberg/JS. Second, we describe the extension to Babelsberg/JS to enable continuous assertions, trigger constraints, and activator constraints. Last, we show how to extend ContextJS layers in order to support scoping of constraints.

6.1 Constraint Construction and Maintenance in Babelsberg

In this section we briefly explain how a constraint is constructed in Babelsberg/JS. As a simple example, suppose we want to specify that the `y` coordinate of a point object `pt` is twice as big as its `x` coordinate. Listing 6.1 shows the constraint. As this constraint is linear, we can use the Cassowary constraint solver to solve the constraint as done in line 1. Line 2 contains the actual constraint expression. Babelsberg/JS employs a preprocessing step to ready the code for execution. This preprocessing step applies a transformation to the source code before its actual execution. The Babelsberg/JS source code transformation emits the appropriate call to the library as seen in Listing 6.2. Additionally, the source code transformation adds a context object to the function call. This object contains all variables that are directly referenced by the function. In order to interpret the constraint expression the interpreter requires this context object.

```
1 always: { solver: new ClSimplexSolver()
2   pt.x * 2 == pt.y
3 }
```

Listing 6.1: A simple constraint before source code transformation

```
1 bbb.always({
2   solver: new ClSimplexSolver()
3   ctx: {
4     pt: pt
5   }
6 }, function() {
7   return pt.x * 2 == pt.y
8 });
```

Listing 6.2: A simple constraint after source code transformation

When calling Babelsberg/JS's `always` method, the given function is delegated to a JavaScript interpreter alongside contextual information. When a constraint is evaluated, the normal execution by the interpreter is adapted using a layer, the `ConstraintConstructionLayer`. In order

to intercept accesses and assignments to relevant variables later during execution, interpretation modified by this layer wraps properties with property accessors. For example, to get the value of a variable, the accessors returns the appropriate value from a constraint solver. However, during constraint construction Babelsberg/JS does not use these accessors but returns a `ConstrainedVariable` object instead. To get a specialized `ConstrainedVariable` for a concrete solver, its `constrainedVariableFor` method is invoked. The solver returns a `ConstrainedVariable` if the interpreter tries to access a value of a type supported by the specific solver. For instance, when accessing the `x` coordinate of the point object, Cassowary returns a `ConstrainedVariable` as a stand-in for the `Number` value. Any messages are sent to this object instead of calculating values. Each `ConstrainedVariable` has a set of supported methods. These methods are used to create primitive constraint objects that can be satisfied using the solver. As a consequence, the interpreted constraint expression returns a specific constraint object that supports the `enable` and `disable` methods. When a constraint was successfully created, this constraint is immediately enabled and solved.

Besides constructing and solving a constraint once, another important feature of an `always` constraint is the automatic maintenance of the desired relation. To do so, the installed property accessors intercept when a new value is assigned to a property wrapped with a property accessor. In this case, the `suggestValue` method of the corresponding `ConstrainedVariable` object is called. To solve the assignment Babelsberg creates a temporary equality constraint between the variable and its new value. However, this change could affect the whole constraint system. Babelsberg finds all constraints with the modified variable. Babelsberg repeats this process for all variables of all constraints found in the previous step, and so on. This process stops once the convex hull of the assignment is found. In order to solve the constraints, Babelsberg sorts all participating solvers according to their weight. Then, the solver with the highest weight solves all of its constraints using its specialized solving algorithm. Babelsberg continues to solve the constraint system with the next solver. However, all variables referenced by previous solvers are marked as read-only. As a consequence, a solver cannot modify variables assigned by previous solvers. This process continues for all remaining solvers. As a result, the constraint system is satisfied and the regular program execution can proceed with the next statement.

Limitations of the current implementation. Babelsberg/JS is an implementation of the object constraint programming (OCP) paradigm provided as a library without any VM support. Such an implementation has several requirements on the host language [12]. One of those requirements is that the host language must support means to intercept variable lookup. Babelsberg/JS uses property accessors to meet this requirement. However, this technique only supports object fields, and cannot intercept local variables. Additionally, some fields of built-in types, such as the `length` attribute of `Arrays`, do not support property accessors. Another requirement is that the host language must provide means to modify the interpretation of a block. As described in this section, Babelsberg/JS uses an augmented JavaScript interpreter for constraint construction. Yet, this interpreter cannot access any variable scoped in a closure. As a consequence, Babelsberg/JS cannot instrument any method refined by ContextJS, because ContextJS employs scoped variables. In addition, current state of the art, browser-based JavaScript modules, asynchronous module definitions, cannot be instrumented if the module definition uses locally scoped variables.

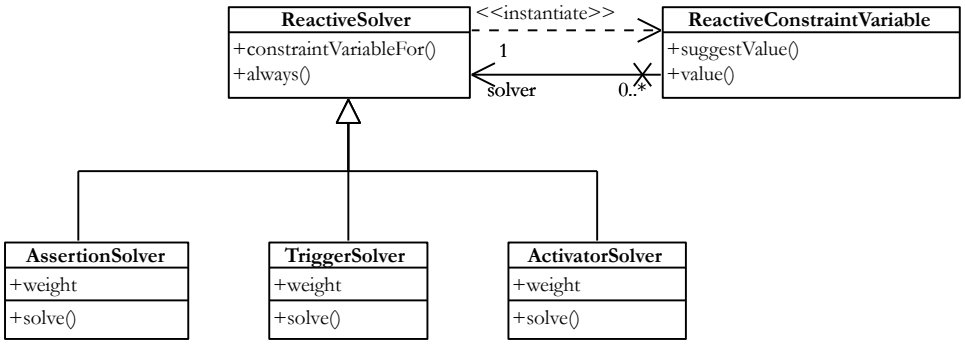


Figure 6.1: Hierarchy of solvers representing additional concepts

6.2 Utilizing the Architecture of Cooperating Solvers to Implement Additional Concepts

In order to implement the concepts described in Chapter 5, we want to extend the existing Babelsberg/JS library. However, we want to apply the least possible changes to Babelsberg/JS. To do so, we search for points of variation in Babelsberg/JS. We identified constraint solvers as such a point, because Babelsberg is meant to be extended using additional solvers. To be specific, we can utilize the architecture of cooperating solvers to implement the concepts as additional solvers. Doing so allows to decouple the existing Babelsberg/JS from the additional concepts and requires only minor changes to Babelsberg/JS itself. Additionally, the implementation as solvers allows an easy integration with existing behavior, because both use the same techniques. The described concepts, continuous assertions, trigger constraints, and activator constraints, share the same domain of arbitrary objects and primitive types they work on. They only differ in their solving behavior. Accordingly, we implement a shared solver superclass, `ReactiveSolver`, for all these concepts. This superclass deals with the necessary communication with Babelsberg/JS. As depicted by Figure 6.1, each concept is provided by a solver class that inherits from `ReactiveSolver` and implements its own means to fulfill the desired property.

When one of the constructs is used, a new solver instance of the respective class is created, e.g. a `TriggerSolver`. Then, Babelsberg/JS's `always` method is invoked, and instructs the interpretation of the given constraint expression using available context information and the solver instance. When accessing a field upon interpretation, Babelsberg/JS calls the `constraintVariableFor` method of the given solver as explained in Section 6.1. The `ReactiveSolver` returns a `ReactiveConstraintVariable` for every object and primitive value, regardless the type of the accessed value. As a consequence, arbitrary constraint expressions are supported. In contrast, ordinary solvers only construct constraint variables for particular types and values, e.g. Cassowary only supports `Number` values. Additionally, a Cassowary constraint variable only responds to certain operations such as +, ∗, and ==. On the contrary, `ReactiveConstraintVariables` do not respond to any operation, instead, the logic is passed back to the Babelsberg/JS interpreter. However, constructing a constraint variable for each field access also allows simple constraint expressions over field accesses that do not involve any calculation. As a result of the interpretation, the Babelsberg/JS constraint is created and all property accessors delegate

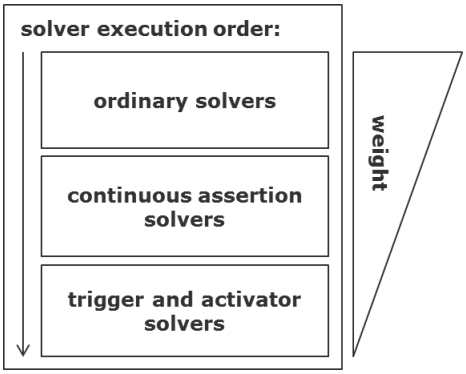


Figure 6.2: Execution order to solve constraints depends on the solver's weight

to an appropriate `ReactiveConstraintVariable`. One has to keep in mind that Babelsberg/JS expects each constraint expression to evaluate to true. This is typically enforced by the `always` statement. Yet, trigger and activator constraints do not require the expression to evaluate to true. Babelsberg/JS considers a constraint expression that evaluates to false as an unsatisfiable constraint and raises an error. Therefore, we added the option `allowFailing` for this very case. This option is automatically set when using our solvers.

Aside from constructing constraints, the other main task of Babelsberg is to maintain the desired properties. Therefore, the `solve` method of a solver is called when a constraint is enabled, usually immediately after its construction, or a `ConstrainedVariable` is modified. Both cases invoke the `solve` method that is implemented for each concrete solver. Figure 6.2 depicts the order in which the additional concepts are executed in relation to ordinary solvers. The order of execution is determined by the weight of the solvers in descending order. First, Babelsberg/JS processes ordinary solvers. They maintain the user-defined constraint system. Next, Babelsberg/JS checks all continuous assertions. Finally, trigger and activator constraints may invoke or adapt behavior depending on the evaluation result of their constraint expressions. To ensure this particular order, `AssertSolvers` have a lower weight than any ordinary solver, and `TriggerSolver` and `ActivatorSolver` have an even lower weight.

Continuous Assertion Solver. A continuous assertion prevents the program from entering an erroneous state as described in Section 5.1. To do so, the solver checks whether the given predicate is fulfilled as illustrated in line 2 of Listing 6.3. If that is not the case, the solver throws a specialized error as seen in line 3. This error causes Babelsberg/JS to break out of its solving routine. Canceling the solving routine causes all downstream trigger and activator constraints to be omitted as the failed assertion detects that the program has entered an inconsistent state. Additionally, the error instructs the `ConstraintVariable` to revert to its previous value, or disables the constraint that causes the error. Currently, reverting the state is limited to assignments which suffices in basic cases.

Trigger Solver. Trigger constraints promise that a given piece of behavior will be invoked as soon as a given condition becomes true. To do so, the solver keeps track of whether the predicate was fulfilled during the last evaluation as seen in line 7 of Listing 6.4. Initially, the solver's `previoslyFulfilled` attribute is set to false. As specified in line 4, if the given predicate is now

```
1  solve: function() {
2    if(!this.constraint.predicate()) {
3      throw new ContinuousAssertError(this.message);
4    }
5  }
```

Listing 6.3: The `solve` method of the `AssertSolver`

fulfilled but was not during last evaluation, the given function should be invoked. To invoke a function, Babelsberg/JS adds the function to a list of callbacks as in line 5. These callbacks are called at the end of the entire solving routine. So, callbacks can modify `ConstraintVariables` and, in turn, cause Babelsberg/JS to solve constraints again.

```
1  solve: function() {
2    var predicateFulfilled = this.constraint.predicate();
3
4    if(predicateFulfilled && !this.previouslyFulfilled) {
5      bbb.addCallback(this.callback, this.constraint.bbbConstraint, []);
6    }
7    this.previouslyFulfilled = predicateFulfilled;
8  },
```

Listing 6.4: The `solve` method of the `TriggerSolver`

Activator Solver. Activator constraints apply a certain adaption in behavior as long as a given condition holds. To do so, the solver checks whether the evaluation result of the condition coincides with the layer activation status. So, if the condition is fulfilled but the layer is currently deactivated, the solver activates the layer as seen in lines 5 and 6 of Listing 6.5. In the same way, if the layer is active but the condition evaluates to false, the solver deactivates the layer as stated in lines 7 and 8.

```
1  solve: function() {
2    var layerIsGlobal = this.layer.isGlobal(),
3        predicateFulfilled = this.constraint.predicate();
4
5    if(predicateFulfilled && !layerIsGlobal) {
6      this.layer.beGlobal();
7    } else if(!predicateFulfilled && layerIsGlobal) {
8      this.layer.beNotGlobal();
9    }
10 },
```

Listing 6.5: The `solve` method of the `ActivatorSolver`

An alternative implementation strategy. Another way to implement the described concepts without altering the Babelsberg/JS library at all is to implement a separate library. However, this new library also has to employ a JavaScript interpreter like Babelsberg/JS does. This results in code duplication. Additionally, the interaction between ordinary constraints and our new concepts would suffer. For instance, the execution order of both libraries might depend on the order of specification of the concepts. To be specific, consider a trigger constraint and an always statement with the same constraint expression. The correct execution order of the solver and

the trigger is crucial if an assignment invalidates the constraint expression. If the trigger is processed first, the assignment would reset the trigger. Next, the always constraint resatisfies the constraint expression. Then, the trigger invokes its callback, because the constraint expression evaluates to true again. In contrast, if the always constraint is processed first, the constraint expression would be resatisfied immediately. As a consequence, the trigger is not reset, thus, no callback is invoked. We argue that implementing the additional concepts as separate library would lead to less predictable behavior as exemplified.

6.3 Using ContextJS Layer to Scope Constraints

Scoped constraints are only enabled as long as an associated layer is active. The implementation of scoped constraints is twofold. First, we need to provide means to create constraints associated with a layer. Second, constraints have to be enabled and disabled when the associated layer is activated and deactivated, respectively.

To create a constraint associated with a layer, we added the method `predicate` to the class `Layer`. This method creates and returns a `LayeredPredicate` as seen in line 3 of Listing 6.6. A `LayeredPredicate` contains all information that is necessary to create a `Constraint` just like ordinary `Predicates` as explained in Section 5.4. Additionally, the associated `Layer` is stored as shown in line 4 of Listing 6.7.

```
1  Object.extend(Layer.prototype, {
2    predicate: function(func, opts) {
3      return new LayeredPredicate(func, opts, this);
4    },
5    _enableConstraints: function() {
6      this._constraintObjects.forEach(function(cobj) {
7        cobj.enable();
8      });
9    },
10   _disableConstraints: function() {
11     this._constraintObjects.forEach(function(cobj) {
12       cobj.disable();
13     });
14   }
15 });
```

Listing 6.6: Extention of the ContextJS `Layer` class

```
1  Predicate.subclass("LayeredPredicate", {
2    initialize: function($super, func, opts, layer) {
3      $super(func, opts);
4      this.layer = layer;
5    },
6    always: function($super, opts) {
7      opts.postponeEnabling = !this.layer.isGlobal();
8      var cobj = $super(opts);
9      this.layer._constraintObjects.push(cobj);
10     return cobj;
11   }
12 });
```

Listing 6.7: Parts of the `LayeredPredicate` class definition

A `Predicate` provides methods to create actual constraint objects. The lines 6 to 11 exemplify this for the method `always`. First, line 7 sets the `postponeEnabling` attribute in the options object depending on whether the associated layer is active. We added the `postponeEnabling` option to be able to prevent Babelsberg/JS from immediately enabling newly created constraints. Then, the constraint is constructed in the usual manner in line 8. Thereby, the constraint is only enabled if the layer is already active. Next, we add the constraint to a list of constraints associated with the layer as shown in line 9. Finally, we return the created constraint. To sum up, the method creates a constraint and associates it with the layer.

The second portion of the implementation regards the dynamic adaption of constraints when the activation state the associated layer changes. To do so, we extend the activation methods of ContextJS. For instance, Listing 6.8 shows how the global activation of a layer is extended. First, we call the layer's `_enableConstraints` method. This method enables all constraint objects associated with the layer as shown in lines 5 to 9 in Listing 6.6. Then, we proceed with the original function. The methods for global deactivation as well as control flow-based activation are extended in the same way.

```
1 cop.enableLayer = cop.enableLayer.wrap(function(callOriginal, layer) {
2   layer._enableConstraints();
3   return callOriginal(layer);
4 });
```

Listing 6.8: Global layer activation extended to activate associated constraints

7 Evaluation

This chapter evaluates the appropriateness of the concepts proposed in Chapter 5 for the sample application described in Section 4.1. Additionally, we present alternative solution strategies and discuss advantages and disadvantages regarding our concepts. Each section of this chapter is structured as follows. First, we describe a concrete challenge in the implementation of the example game. Next, we present an implementation of this particular problem using one or more of our proposed concepts. Then, we describe an alternative implementation using other appropriate concepts. Finally, we compare both implementations.

Note, that we do not evaluate the two following concepts in this chapter. First, we do not discuss the usage of regular constraints in this chapter. Section 4.4 describes how we use DeltaBlue constraints as a high-level abstraction of traditional data flow mechanisms in our sample application. Second, we do not evaluate continuous assertions. Continuous assertions allow to deal with otherwise unsolvable constraints in certain, restricted situations. Section 5.1 discusses two possible alternatives: to develop a dedicated solver for each of such problems, or to use scattered code fragments to maintain the desired properties. While the first option presents an unnecessary overload especially for object-oriented (OO) developer, we lose all advantages of constraint programming (CP) using the second option.

7.1 Collision Detection using Trigger Constraints

Problem. The example game features three different types of entities: tanks, bullets, and power ups. In the process of the game, entities move and eventually collide with each other. When that happens, an entity-dependent collision resolution should be invoked, e.g. a bullet colliding with a tank destroys both entities. To be specific, the problem is to implement a mechanism for collision detection and then apply a given collision resolution procedure in response.

We choose collision detection and resolution as an example for complex interactions of game objects and their environment. Especially in game development objects frequently have to react to a certain state of the program.

Solution. To handle arbitrary responses to collisions the `GameObject` provides the `onCollisionWith` method as seen in Listing 7.1. The method is called on a `Gameobject` with two parameters: a second `GameObject` and a response to the collision of both. To implement this method we use a constraint expression that specifies whether both objects overlap as seen in lines 4 to 6. Then, we create a trigger constraint that invokes the given callback once the constraint expression evaluates to true in lines 6 to 8.

Alternative. Alternatively, we could make use of a topic-based messaging system which is a frequently used approach in game development. In a topic-based messaging system objects

```
1  onCollisionWith: function(other, callback) {
2    var that = this;
3
4    return predicate(function() {
5      return that.position.distance(other.position) <= that.radius + other.radius;
6    }).trigger(function() {
7      callback.call(this, that, other);
8    });
9  },
```

Listing 7.1: Method `onCollisionWith` to specify how a collision between two `GameObjects` is handled

can subscribe callbacks to a topic of their interest. Additionally, objects can emit concrete messages related to a topic. As a consequence, all subscribed callbacks are invoked with the specific message as parameters.

Applied to our example, this solution requires a physics component that checks for collisions during the execution of the game loop. As seen in lines 4 to 12 in Listing 7.2, the method `checkCollisions` iterates twice over all game objects and checks for a collision of each two game objects. If a collision occurs, the corresponding objects are stored. Then, for each stored collision we emit a message of the appropriate topic as shown in lines 14 to 16. Listing 7.3 shows the implementation of the `onCollisionWith` method in this variant. As seen in line 4, we subscribe to the appropriate topic. However, we have to check whether the colliding objects are the objects of interest before invoking the given callback as shown in line 5.

```
1  checkCollisions: function() {
2    var collisions = [];
3
4    this.gameObjects.each(function(object1, index1) {
5      this.gameObjects.each(function(object2, index2) {
6        if(index1 <= index2) { return; }
7        if(object1.position.distance(object2.position) <= object1.radius + object2.radius) {
8          collisions.push([object1, object2]);
9          collisions.push([object2, object1]);
10       }
11     });
12   });
13
14   collisions.each(function(collision) {
15     topic('collision').emit(collision);
16   });
17 },
```

Listing 7.2: Collision detection in a topic-based messaging approach

```
1  onCollisionWith: function(other, callback) {
2    var that = this;
3
4    topic('collision').subscribe(function(object1, object2) {
5      if(object1 === that && object2 === other) {
6        callback(that, other);
7      }
8    });
9  },
```

Listing 7.3: Method `onCollisionWith` using a topic-based messaging approach

Comparison. By comparing both variants we discover two major advantages of trigger constraints over the messaging system. First, trigger constraints allow to specify the reactive behavior explicitly. On the contrary, the messaging system uses an additional layer of indirection, the topic. Topics are a powerful decoupling mechanism, yet the connection between cause and effect becomes less clear. Second, the detection mechanism of the messaging system variant is limited. To be specific, this variant checks the condition only once per frame. Suppose a tank pushes another tank into a power up as a response to the collision of both tanks. The collision with the power up is not handled until the next frame. In contrast, triggers automatically intercept every statement that may change the condition. As a consequence, trigger constraints invoke the callback correctly as soon as the given condition becomes true.

7.2 Power Ups using Layers and Activator Constraints

Problem. Power ups are one of the three entity types in our example game. When a tank collects a power up by touching it, the power up enhances the tank's abilities for a limited time. ContextJS layers provide a convenient way to express behavioral variation. The problem, however, is to find an appropriate activation mechanism to dynamically activate the layer in this scenario.

Power ups represent an example of context-dependent behavior variation. Thereby, contexts are frequently defined by a condition such as in this example.

Solution. Figure 7.1 shows the architecture we use to handle power ups and their temporal benefits. A separate class, the `Collectible`, handles the appearance and physics of a power up. When a collision with a tank occurs, the `Collectible` is removed from the level and activates the `PowerUp`. An activated `PowerUp` creates a new `Timer` for the combination of tank and power up type , or resets a `Timer` if already existent. Listing 7.4 shows the `Timer` class which measures time intervals. As seen in lines 6 to 8, in its constructor the `Timer` creates a constraint expression that indicates whether a timeout occurred or not. Because we handle constraints as first-class entities, the `PowerUp` can use this `untilTimeout` constraint expression provided by the `Timer`. Next, this constraint expression is used to activate the behavior variation of the `PowerUp` as seen in Listing 7.5. The effect is implemented in each subclass of `PowerUp` using a layer to refine the tank's behavior. Listing 7.6 exemplifies the effect of the `SpringPowerUp` which allows fired `Bullets` to ricochet one additional time.

Alternative. Another possibility to implement the desired feature without altering the generic `Timer` class is to use mixins. Similar to FlightJS[1] we use functional mixins to extend objects in an aspect-oriented manner. We discuss the relation between aspect-oriented programming (AOP) and trigger constraints in more detail in Section 8.1. Listing 7.7 uses functional mixins to extend the `timer` object in order to activate the power up layer appropriately. First, we activate the layer globally (line 12). Then, the `withAdvice` mixin extends the `timer` with `before` and `after` methods (line 13). Next, the `withLayerUntilTimeout` mixin extends the methods of the `timer`. After the `update` method is called, we check whether a timeout has occurred and if the layer is currently active (lines 2 - 6). If that is the case, we deactivate the layer. After the `reset` method ran, we activate the layer again (lines 7 - 9).

[1] Twitter Inc., FlightJS, `http://flightjs.github.io/` (last accessed April 30, 2015)

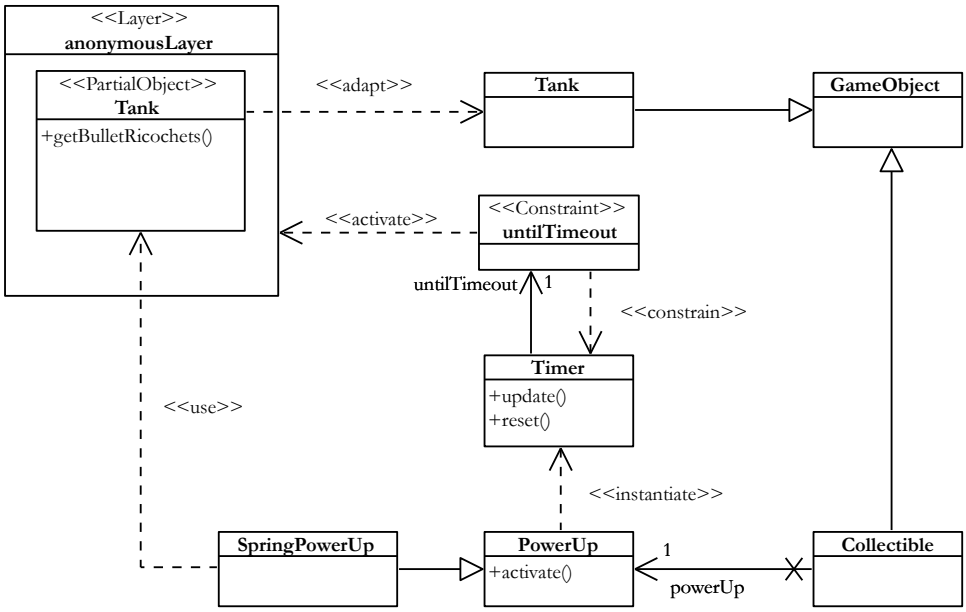


Figure 7.1: Architecture to handle behavior variations of power ups

```
Object.subclass('Timer', {
  initialize: function(time) {
    var that = this;

    this._time = time;
    this.untilTimeout = predicate(function() {
      return that._time > 0;
    });
  },
  update: function(dt) {
    this._time -= dt;
  },
  reset: function(additionalDuration) {
    this._time = Math.max(this._time, 0) + additionalDuration;
  }
});
```

Listing 7.4: `Timer` class for measuring time intervals

```
timer.untilTimeout.activate(this.effect(tank));
```

Listing 7.5: Connecting the timer with the power up-specific effect

```
effect: function(tank) {
  return new Layer().refineObject(tank, {
    getBulletRicochets: function() {
      return cop.proceed() + 1;
    }
  });
},
```

Listing 7.6: The `effect` method of the class `SpringPowerUp`

```
1  function withLayerUntilTimeout() {
2    this.after('update', function() {
3      if(this.time <= 0 && layer.isGlobal()) {
4        layer.beNotGlobal();
5      }
6    });
7    this.after('reset', function() {
8      layer.beGlobal();
9    });
10 }
11
12 layer.beGlobal();
13 withAdvice.call(timer);
14 withLayerUntilTimeout.call(timer);
```

Listing 7.7: An equivalent implementation using aspects

Comparison. Using activator constraints, one can bind the activation of a layer to an arbitrary condition. We argue that this mechanism has a number of advantages compared to the mixin variant. First, activator constraints make the relation between the condition and the layer activation explicit. In contrast, the mixin variant treats this relation implicitly by adding the appropriate condition in the extended functionality as seen in line 3 of Listing 7.7. Second, using mixins one has to explicitly specify which methods can affect the layer. So, this variant requires knowledge about the base object. Additionally, when the base object is changed, the mixin has to be updated as well. On the contrary, activator constraints handle this issue automatically by installing property accessors at the appropriate places. Nevertheless, functional mixins represent a powerful composition mechanism in JavaScript.

7.3 Handle Constraints in Different Game Modes

Problem. The example game supports two operational modes: the game mode and the editor mode. Both modes vary in their specific behavior. In addition to varying behavior, several regular constraints and trigger constraints apply only in one of the modes. The problem is to find a mechanism to deal with the varying behavior as well as varying constraints in both modes.

The modes exemplify context-dependent constraints. We argue that a powerful scoping mechanism for both, behavioral variation and constraints, is desirable in an object constraint programming (OCP) environment.

Solution. Layers provide a convenient scoping mechanism for behavioral variation. For instance, the `EditorLayer` refines the `Game`'s `draw` method as shown in Listing 7.8. The rendering of the level is adjusted to draw the level semitransparent (line 4). Additionally, the interface of the editor mode is drawn (lines 6 - 8). The addition of scoped constraints described in Section 5.3 allows layers to treat constraints in a similar way as variations in behavior. So, we can specify that certain constraints are only enabled while the associated layer is active. Listing 7.9 exemplifies this for two constraints which handle editor mode-specific functionality.

Alternative. Without scoped constraints we have to enable or disable constraints using low-level methods as layers activate or deactivate, respectively. Some context-oriented program-

```
1  EditorLayer.refineClass(Game, {
2    // ...
3    draw: function() {
4      this.renderer.withTransparency(0.7, cop.proceed.bind(cop));
5
6      this.renderer.withViewport(this.viewport, (function() {
7        this.editor.draw(this.renderer);
8      }).bind(this));
9    }
10 });
```

Listing 7.8: The `EditorLayer` augments the `Game`'s `draw` method

```
1  EditorLayer.predicate(function() {
2    return input.pressed('leftclick');
3  }).trigger(
4    editor.modifyTileType.bind(editor)
5  );
6
7  EditorLayer.predicate(function() {
8    return editor.tileIndex.equals(map.positionToCoordinates(input.position));
9  }).always({
10   solver: new DBPlanner()
11 });
```

Listing 7.9: Two constraints that are only active in editor mode

ming (COP) implementations[2] allow to bind callbacks to the activation and deactivation of layers in order to support the setup and cleanup for additional behavior. We extend ContextJS to support this feature, and use callbacks to enable and disable context-dependent constraints appropriately. Listing 7.10 shows a trigger constraint in lines 1 to 5. In line 7 we bind the `enable` method of the trigger constraint to the activation of the `EditorLayer`. Similarly, we disable the constraint again if the layer is deactivated (line 8). As a result, we achieve a behavior similar to scoped constraints.

```
1  var modifyTileTrigger = predicate(function() {
2    return input.pressed('leftclick');
3  }).trigger(
4    editor.modifyTileType.bind(editor)
5  );
6
7  EditorLayer.on('activate', modifyTileTrigger.enable.bind(modifyTileTrigger));
8  EditorLayer.on('deactivate', modifyTileTrigger.disable.bind(modifyTileTrigger));
```

Listing 7.10: Constraints scoped implicitly through layer callbacks

Comparison. We argue that scoped constraints have the following two advantages over the event-based implementation. First, scoped constraints make the desired relation between constraint and layer explicit and, therefore, can reflect the requirements in a *what* fashion. In contrast, using layer callbacks and low-level modification methods solves the same issue in a *how* fashion. So, the code reflects the requirement less clearly. Yet, layer callbacks can involve arbitrary behavior and, therefore, represent a powerful mechanism. On the contrary, scoped

[2] Marius Colacioiu, Cop.js, http://www.colmarius.net/cop/ (last accessed April 30, 2015)

constraints represent a special purpose solution which is easier to read, yet limited in its application. Second, scoped constraints allow to treat variations in behavior and constraints equally. In contrast, the alternative implementation has to treat constraints differently from behavioral variation. Thus, the concept of scoped constraints provides an appropriate solution in an OCP environment.

Note, that scoped constraints as well as variations in behavior rely on the underlying COP implementation and its layer activation mechanisms. As we support activators, constraints can be bound to arbitrary conditions.

8 Related Approaches

This thesis describes an extension to the Babelsberg design to better integrate constraints with object-oriented programming (OOP) environments. Thereby, we focus on concepts to adapt or invoke certain behavior based on given conditions. Several concepts regarding the same issue have been proposed. Those concepts are discussed in this section.

As an additional contribution, we discuss the implementation of a non-trivial application scenario that originates from the domain of OOP. No other work using an integration of constraint programming (CP) and OOP to implement a non-trivial problem that clearly originates from the field of OOP is known to us.

8.1 Trigger Constraints in Relation to Aspect-Oriented Programming

The concept of aspect-oriented programming (AOP) [27] is a technique for improving separation of concerns in software. Aspects are modular units of crosscutting implementation, comprised of three concepts: *join points*, *pointcuts*, and *advices*. Join points are points in the execution of the program such as a method call or a field access. Pointcuts are collections of join points. Advices are partial methods that can be attached to pointcuts. Once the execution of a program reaches a specified join point, the given advice is executed. This mechanism allows to easily modularize crosscutting concerns of a program such as logging.

Both concepts, aspects and trigger constraints, allow to invoke arbitrary callbacks at specific points during the execution of a program. So, the callback of a trigger is analogous to an aspect's advice. Yet, using AOP, one has to explicitly specify the very points to intercept the execution of the program. In contrast, trigger constraints allow to define those points implicitly by specifying an arbitrary condition.

8.2 Activator Constraints in Relation to Other Layer Activation Mechanisms

In this thesis, we identified the need to scope constraints, similar to behavioral variation, to fit the requirements of object-oriented (OO) environments. The context-oriented programming (COP) paradigm provides a novel approach to adapt behavior depending on the current context. We make use of layers to scope constraints as well. To our knowledge this is the first integration of layers and constraints. In the following we compare activator constraints with other layer activation mechanisms.

Event-based activation. Typical layer activation mechanisms such as imperative and control flow-based activation tend to couple activation logic and base program logic. In contrast, JCop [3] separates the control of layer activation, and the execution of context-dependent behavior. To do so, JCop introduces a *declarative layer composition* statement that consists of

two parts: *predicates* define a set of events provided by the system and a *composition block* specifies which layers should be activated or deactivated. When one of the specified events fires, the corresponding layers are activated or deactivated, respectively. Another COP implementation, EventCJ [24], provides two new language constructs, *event declarations* and *layer transition rules*, in addition to layer declaration. An event declaration specifies a named event, when the event is triggered, and to which objects the event is sent. A layer transition rule specifies the activation and deactivation of layers in a declarative manner. If an object receives the specified event, the rule is applied to this very object. For instance, such a rule could state that the object switches from one set of layers to another.

Similar to activator constraints, the event-based activation mechanisms of JCop and EventCJ allow to decouple the activation of layers and the base program by abstracting the point of activation. To do so, JCop and EventCJ events rely on a subset of AspectJ [26] pointcuts. However, the pointcut has to be specified explicitly. In contrast, activator constraints allow to specify an arbitrary condition and, thereby, abstract from concrete point of execution to activate the layer. Additionally, while activator constraints activate a layer globally, EventCJ provides layer activation in an instance-specific manner.

Implicit activation. In most existing implementations of method layers, a layer has to be activated explicitly, typically after evaluating some condition. If this condition could change frequently, the check and the layer activation need to be added in many places of the code. To address this issue, PyContext [38] introduces implicit layer activation. Each PyContext layer may define the `active` method. If this method evaluates to true, the layer is active. When a layered method is called, PyContext first determines which layers are active. Then, the layer composition for all participating method definitions is built. Another COP implementation, ServalCJ [25], attempts to unify all existing layer activation methods by introducing two concepts: *contexts* specify the duration of a layer activation and *subscribers* specify which computations are affected by the layer activation. The unified model allows to represent all existing COP implementations. This model is based on several activation means, including the same concept of implicit layer activation as PyContext.

Implicit layer activation allows to factor out context activation from the actual logic of the program like activator constraints do. However, one should keep in mind that PyContext as well as ServalCJ evaluate the activation condition on each invocation of a method that is potentially affected by layer activation. This is an appropriate mechanism, if a condition changes more often than the affected method is invoked. If that is not the case, this mechanism may produce severe overhead. In contrast, activator constraints only check conditions at the moment they could change, i.e. during an assignment or during constraint solving. As an example, in our application the user switches between game mode and editor mode by pressing a button. The mode affects the behavior of the `game` object that is called every frame. PyContext would check the condition on every frame, Babelsberg only when the button state changes.

8.3 Reactive Programming

Trigger and activator constraints allow to write an application in a reactive manner. Reactive programming promises clean, declarative code and, therefore, has been of interest for researchers. Two specific approaches in reactive programming are discussed in the following.

Functional reactive animation. Fran [10] is a collection of data types and functions dedicated to create richly interactive, multimedia applications in Haskell. To achieve this goal, Fran relies on three basic concepts: *behaviors*, *events*, and *declarative reactivity*. First, behaviors are values that vary over continuous time. One can specify such behaviors in relation to the built-in `time` variable or to other behaviors. Second, events are named entities described by arbitrary complex predicates. Third, Fran allows to react to events in a declarative manner. These reactions are composed using events and temporal logic.

With events Fran offers a similar concept as trigger constraints, both allow the specification of arbitrary complex conditions. However, the underlying mechanisms of both concepts are quite different. To detect events over continuous time Fran employs interval analysis. In contrast, trigger constraints react to any discrete, imperative state changes. By introducing reactivity to functional programming, Fran laid the foundation of the concept of functional reactive programming. We argue that triggers represent a powerful tool to specify reactivity in OO systems.

Constraint reactive programming. The Babelsberg/CRP design attempts to integrate object constraint programming (OCP) with reactivity. To do so, Babelsberg/CRP draws heavily from the concepts of Fran. To be specific, Babelsberg/CRP is based on continuously varying variables and boolean-valued event variables that are similar to Fran's behaviors and events, respectively. Babelsberg/CRP introduces two types of temporal constraints: a *while* constraint continuously enforces a set of constraints as long as the given predicate holds and a *when* constraint instantaneously enforces a set of constraints when an event occurs. Both types of constraints allow for an *until* clause with an event expression that cancels the constraint if the event occurs.

The while temporal constraint act in a similar way as activator constraints. However, the *while* constraint is only able to scope other constraints. In contrast, activator constraints can activate layers, and these layers can adapt constraints and behavior. The *when* temporal constraint and the *until* clause are both similar to trigger constraints. Yet, Babelsberg/CRP and our reactive constraints greatly differ in their approaches. Similar to Fran, Babelsberg/CRP reacts to changes in a continuous environment. On the contrary, reactive constraints as introduced in this work integrate into discrete, OO environments.

9 Future Work

We presented a non-trivial application scenario to verify Babelsberg's capabilities and applicability to classical object-oriented (OO) problems. We have shown some fields of applicability of Babelsberg, but also identified shortcomings in the current Babelsberg design. To address these issues, we extended Babelsberg with the concept of reactive and scoped constraints. Nevertheless, we identified some further areas of improvement. For example, we did not improve the usability of the existing functionality of Babelsberg.

9.1 Extending Babelsberg Functionality

Integrating a general purpose solver. One major contribution of constraint programming (CP) is that we are able to solve hard problems like layouting using an appropriate solver, e.g. Cassowary. Babelsberg already supports the integration of additional solvers in order to cover a wider range of domains. However, the available solvers are typically limited to specific, primitive domains, as described by Wallace [39]. As a consequence, most constraints involving high-level objects cannot be solved using Babelsberg. To increase the applicability of object constraint programming (OCP), the development and integration of solvers that address complex structures like object graphs seems desirable. One way to do so is to involve user-defined functions in the process of constraint solving. For example, the DeltaBlue solver requires the user to supply propagation functions in order to solve given constraints. This requirement puts an additional load on the programmer, but allows the solver to support arbitrary domains. Babelsberg already supports the DeltaBlue solver that is also used in our OO sample application, as described in Section 4.4. Another solver that could increase the applicability of Babelsberg is the van Overveld relaxation solver [32]. This solver requires the programmer to specify an error function to detect whether a constraint is satisfied and a displacement function that is called to correct that error. So, the van Overveld relaxation solver supports any domain because arbitrary functions can be supplied.

Instance-specific layer activation. This thesis introduced constrained layers which are active as long as a particular expression is fulfilled. While constrained layers are useful for conditional variations of behavior, adapting the behavior of specific instances on a condition currently involves creating one layer per instance. The concept of instance-specific layer activation [28] seems useful in those cases. However, this type of layer activation is currently not supported by constrained layers. We imagine a layer to be activated for an instance if a given condition is met for this specific object. Note, that to implement such a behavior, a notion of parameterizable constraints is needed. So, the constraint would be defined once, but checked for each instance of a class.

9.2 Improving Babelsberg Usability

This thesis proposed a possible way to better integrate CP features with OO behavior. However, we investigated further fields of possible improvements. These improvements mainly focus on making Babelsberg easier accessible for OO programmers.

Automatic edit constraints. Some solvers, like DeltaBlue and Cassowary, have a high cost for the initial construction of constraints. As described in Section 2.3, both solvers need exponential time with respect to the number of constraints to prepare the constraint system for solving. Actually solving those systems only needs linear time with respect to the number of constraints. However, assignments to constraint variables are modeled using temporary constraints in Babelsberg. This can lead to a high runtime if such constraint variables are changed frequently. For such cases Babelsberg offers *edit constraints* to deal with this issue. These permanent constraints handle assignments without the need to restructure the whole constraint system. As a consequence, the usage of edit constraints can lead to better performance. Nevertheless, two issues need to be taken into consideration when using edit constraints. First, edit constraints require the user to know which variables act as constraint variables and to directly manipulate these variables. This violates the OO principles of encapsulation and information hiding. Second, edit constraints integrate poorly with external libraries. One needs to extend or adjust the libraries in order to profit from edit constraints. Both issues may be solved by letting Babelsberg or the respective solver detect constraint variables that are frequently modified. For those variables Babelsberg automatically constructs edit constraints without affecting the semantics. As a consequence, the overall performance could be improved transparently.

Automatic solver selection and region management. CP requires much knowledge from a programmer in order to make efficient use of its abstraction mechanisms. For example, a programmer has to know all solver specifics to choose the right tool for the task at hand. As a consequence, OO programmers need to learn about these specifics before they can employ constraints efficiently. Additionally, even if the solver capabilities are familiar to the programmer, managing solver instances manually puts an additional load on the programmer. As a system becomes more complex, choosing an optimal distribution of constraints on different solver instances is difficult. A mechanism to help programmers to manage solver instances seems highly desirable. Currently, Babelsberg/JS supports a list of default solvers. If no solver is explicitly specified for a constraint, Babelsberg/JS tries out which default solver can solve the given constraint, and assigns the constraint to the first matching solver. However, considering a complex system with dynamic constraints a more elaborated mechanism would be useful. This mechanism should not only automatically select an appropriate solver during constraint construction, but also reassign constraints to other solvers or regions when beneficial. To do so, one might want to separate code interpretation and constraint construction using an intermediate representation of the constraint expression. So, one could transform the intermediate representation more easily into a solver-specific constraint expression when the desired type of solver is changed. Once Babelsberg is able to manage solvers and regions automatically, the programmer can finally focus on the constraint itself rather than how the constraint is solved and maintained. Ultimately, automatic solver selection and region management would lower the initial barrier for OO programmer.

10 Conclusions

Babelsberg is an instance of the object constraint programming (OCP) design, and aims to be a practical constraint programming (CP) tool for object-oriented (OO) programmers. In this thesis, we evaluated how well Babelsberg was able to achieve this goal. To that end, we reviewed several examples written using an implementation of Babelsberg. It turned out that all current examples emerge from the area of CP, and therefore cannot convincingly demonstrate how useful Babelsberg is for pure OO applications. Hence, we cannot make any conclusions simply based on those examples. Instead, we formulated an example application with OO roots, an adaption of the game Wii Play/Tanks. This example enabled us to study the usage of constraints in an actual OO program. We identified that the usage of constraints is much more limited compared to the previous, trivially-constrainable examples. Nevertheless, the readability of the OO code can benefit from constraint systems: for example, Babelsberg in conjunction with the local propagation solver DeltaBlue allowed us to describe data flows at a high level of abstraction.

However, we also discovered three major shortcomings in the current design of Babelsberg with regards to the usage in our sample application. First, some constraints involving high-level objects cannot be solved by any available solver. Second, constraints lack of integration with the surrounding application behavior. Third, as constraints are enabled and disabled frequently in our example, a sophisticated way to deal with these dynamic constraints is missing.

To overcome these shortcomings, we adjusted the Babelsberg design in various ways. To address the first shortcoming, we introduce *continuous assertions* that allow to deal with the absence of an appropriate solver given a certain constraint. The constraint cannot be solved automatically, but invalidating the constraint expression of a *continuous assertion* automatically restores the previous state. Regarding the second shortcoming we introduce *trigger constraints* that invoke a callback as soon as a given constraint expression evaluates to true. Similarly, an *activator constraint* allows to dynamically adapt context-oriented programming (COP) layers based on the result of a constraint expression. To address the third shortcoming, we developed a mechanism to associate a constraint with a COP layer. As a result, the constraint is automatically enabled while the associated layer is active. The combination with *activator constraints* allows to automatically enable constraints based on arbitrary boolean expressions.

We implemented the proposed concepts in Babelsberg/JS and illustrate their usage in our example application. In addition, we discussed possible alternatives to the usage of constraints in our application. In particular, we investigated data flows, functional mixins, and a topic-based message system. We argue that the proposed concepts represent a useful addition to Babelsberg and to the object constraint programming paradigm in general.

Bibliography

[1] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. "A comparison of context-oriented programming languages." In: *Proceedings of International Workshop on Context-Oriented Programming (COP)*. 6. ACM. 2009, p. 6.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. "ContextJ: Context-oriented programming with Java." In: *Journal of the Japan Society for Software Science and Technology (JSSST) on Computer Software* 28.1 (2011), pp. 399–419.

[3] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. "Event-specific software composition in context-oriented programming." In: *Proceedings of the 9th International Conference on Software Composition (SC)*. Springer. 2010, pp. 50–65.

[4] Greg J. Badros, Alan Borning, and Peter J. Stuckey. "The Cassowary linear arithmetic constraint solving algorithm." In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 8.4 (2001), pp. 267–306.

[5] Alan Borning. "The programming language aspects of ThingLab, a constraint-oriented simulation laboratory." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 3.4 (1981), pp. 353–387.

[6] Alan Borning, Robert Duisberg, Bjorn N. Freeman-Benson, Axel Kramer, and Michael Woolf. "Constraint hierarchies." In: *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 1987, pp. 48–60.

[7] Pascal Costanza and Robert Hirschfeld. "Language constructs for context-oriented programming: an overview of ContextL." In: *Proceedings of the symposium on Dynamic languages (DLS)*. ACM. 2005, pp. 1–10.

[8] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver." In: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[9] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. *Prolog - the standard: reference manual*. Springer, 1996. ISBN: 978-3540593041.

[10] Conal Elliott and Paul Hudak. "Functional reactive animation." In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Vol. 32. 8. ACM. 1997, pp. 263–273.

[11] Tim Felgentreff, Alan Borning, and Robert Hirschfeld. "Babelsberg: Specifying and solving constraints on object behavior." In: *Journal of Object Technology (JOT)* 13.4 (2014), 1:1–38.

[12] Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. "Babelsberg/JS - A browser-based implementation of an object constraint language." In: *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. Springer, 2014, pp. 411–436.

[13] Tim Felgentreff, Todd Millstein, and Alan Borning. *Developing a formal semantics for Babelsberg: A step-by-step approach*. Tech. rep. Technical Report TR-2014-002a, Viewpoints Research Institute, 2014, pp. 1–60.

[14] Bjorn N. Freeman-Benson. "Kaleidoscope: Mixing objects, constraints, and imperative programming." In: *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications / European Conference on Object-oriented Programming (OOPSLA/ECOOP)*. ACM. 1990, pp. 77–88.

[15] Bjorn N. Freeman-Benson and Alan Borning. "Integrating constraints with an object-oriented language." In: *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. Springer. 1992, pp. 268–286.

[16] Bjorn N. Freeman-Benson and John Maloney. "The DeltaBlue algorithm: an incremental constraint hierarchy solver." In: *Proceedings of the International Phoenix Conference on Computers and Communications*. 1989, pp. 538–542.

[17] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. "An incremental constraint solver." In: *Communications of the ACM (CACM)* 33.1 (1990), pp. 54–63.

[18] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983. ISBN: 0-201-11371-6.

[19] Maria Graber, Tim Felgentreff, Robert Hirschfeld, and Alan Borning. "Solving interactive logic puzzles with object-constraints." In: *Proceedings of the Workshop on Reactive and Event-based Languages and Systems (REBLS)*. 2014.

[20] Martin Grabmüller and Petra Hofstedt. "Turtle: A constraint imperative programming language." In: *Research and Development in Intelligent Systems XX, Proceedings of AI2003*. Springer, 2004, pp. 185–198.

[21] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. "Context-oriented programming." In: *Journal of Object Technology (JOT)* 7.3 (2008), pp. 125–151.

[22] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. "The lively kernel a self-supporting system on a web page." In: *Self-Sustaining Systems (S3)*. Springer, 2008, pp. 31–50.

[23] Joxan Jaffar and Jean-Louis Lassez. "Constraint logic programming." In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 1987, pp. 111–119.

[24] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. "EventCJ: a context-oriented programming language with declarative event-based context transition." In: *Proceedings of the tenth international conference on Aspect-oriented software development (AOSD)*. ACM. 2011, pp. 253–264.

[25] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. "Generalized layer activation mechanism through contexts and subscribers." In: *Proceedings of the 14th International Conference on Modularity (MODULARITY)*. ACM. 2015, pp. 14–28.

[26] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. "An overview of AspectJ." In: *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. Springer, 2001, pp. 327–354.

[27] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-oriented programming." In: *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. Springer, 1997, pp. 220–242.

[28] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. "An open implementation for context-oriented layer composition in ContextJS." In: *Journal of Science of Computer Programming* 76.12 (2011), pp. 1194–1209.

[29] Jens Lincke, Robert Krahn, Dan Ingalls, Marko Roder, and Robert Hirschfeld. "The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content." In: *Proceedings of the 45th Hawaii International Conference on Systems Science (HICSS)*. IEEE. 2012, pp. 693–701.

[30] Gus Lopez, Bjorn N. Freeman-Benson, and Alan Borning. "Implementing constraint imperative programming languages: The Kaleidoscope'93 virtual machine." In: *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM. 1994, pp. 259–271.

[31] Gus Lopez, Bjorn N. Freeman-Benson, and Alan Borning. "Kaleidoscope: A constraint imperative programming language." In: *Constraint Programming*. Ed. by Brian Mayoh, Enn Tyugu, and Jaan Penjam. Vol. 131. NATO ASI Series. Series F: Computer and System Sciences. Springer, 1994, pp. 313–329.

[32] C.W.A.M. van Overveld. "30 Years after Sketchpad: Relaxation of Geometric Constraints Revisited." In: *Centrum voor Wiskunde en Informatica (CWI) Quarterly* 6.4 (1993), pp. 363–383.

[33] François Pachet and Pierre Roy. "Integrating constraint satisfaction techniques with complex object structures." In: *Proceedings of the 15th Annual Conference of the BCS Specialist Group on Expert Systems*. 1995, p. 11.

[34] François Pachet and Pierre Roy. "Mixing constraints and objects: A case study in automatic harmonization." In: *Proceedings of TOOLS Europe*. Vol. 95. Prentice Hall. 1995, pp. 119–126.

[35] Pierre Roy and François Pachet. "Reifying constraint satisfaction in Smalltalk." In: *Journal of Object-Oriented Programming (JOOP)* 10.4 (1997), pp. 43–51.

[36] Erica Sadun. *IOS Auto Layout Demystified*. Addison-Wesley, 2013. ISBN: 978-0321967190.

[37] Ivan E. Sutherland. "Sketchpad: A man-machine graphical communication system." In: *Proceedings of the Spring Joint Computer Conference*. AFIPS '63 (Spring). 1963, pp. 329–346.

[38] Martin Von Löwis, Marcus Denker, and Oscar Nierstrasz. "Context-oriented programming: beyond layers." In: *Proceedings of the International Conference on Dynamic Languages (ICDL)*. Vol. 286. ACM. 2007, pp. 143–156.

[39] Mark Wallace. "Practical applications of constraint programming." In: *Constraints* 1.1-2 (1996), pp. 139–168.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst sowie keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe.

Potsdam, den 30. April 2015

Stefan Lehmann