Connecting Object Constraints with Context-oriented Programming

Scoping Constraints with Layers and Activating Layers with Constraints

Stefan Lehmann Tim Felgentreff Robert Hirschfeld

Hasso Plattner Institute, University of Potsdam, Germany

{firstname.lastname}@hpi.uni-potsdam.de

Abstract

Context-oriented Programming extends object-oriented languages with a mechanism to dynamically adapt behavior. Object Constraint Programming orthogonally extends object-oriented run-times by integrating constraints, including support for constraints over mutable state, object identity, and the results of message sends. Using these two language extensions in conjunctions offers interesting opportunities.

In this paper, we report on new mechanisms involving layers and constraints that evolved from our experiences with combining Babelsberg/JS, an Object Constraint Language, and ContextJS, an implementation of Context-oriented Programming in JavaScript. First, our experience shows that it is desirable to dynamically adapt declarative constraints, which offer an orthogonal mechanism to the definition of imperative behavior, likewise at runtime. In this work, we show an extension to ContextJS to scope activation or refinement of constraints dynamically using layers. Second, ContextJS already provides different activation mechanisms for layers, including dynamically or structurally scoped, or globally through system generated events. Constraints provide an activation mechanism based on arbitrary boolean expressions changing their value, allowing for interesting applications of behavioral adaption based on certain conditions.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Constraint Layers, Object Constraint Programming, Context-oriented Programming

COP '15, July 05 2015, Prague, Czech Republic.

ACM 978-1-4503-3654-3/15/07…$15.00.

http://dx.doi.org/10.1145/10.1145/2786545.2786549

1. Introduction

Many mechanisms extend the toolbox of the object-oriented programmer with features such as pattern matching[6], multi-dimensional dispatch[12], and more exist. Among those mechanisms are Context-oriented Programming (COP)[3, 7], which adds dynamic adaption of behavior based on layers, and Object Constraint Programming (OCP)[4, 5], which adds constraints over mutable objects and on the results of message sends. Using these two language extensions in conjunctions offers interesting opportunities to address shortcomings in each language, and thus make both extensions useful for a wider variety of problems. In this paper, we present two such extensions, which we call *Layer Activators* and *Constraint Layers*.

Layer Activators COP layers adapt behavior at runtime when they are activated. There are various activation mechanisms in COP languages, including dynamically scoped[7] or scoped to object structure[11] and event-based[1, 8] or implicit activation[14]. While the first two mechanisms offer explicit control over the activation time of layers, the latter two are particularly interesting for our discussion because of their declarative nature.

Event-based activation uses system-generated events to activate or deactivate a layer during the event loop. Reactive activation is based on similar principles as Aspect-oriented Programming: whenever a message is sent that matches a *pointcut*, an associated condition is evaluated. If that condition evaluates to true, a layer is activated for the duration of the method activation.

A limiting factor of both mechanisms is their granularity. Event-based activation relies on the creating appropriate events that indicate a change in the system state to activate layers. In JavaScript, for example, common system generated events indicate user input or network activity. However, it is not possible to tell the system to generate an event whenever an arbitrary condition changes. Similarly, reactive activation focuses on message sends. The conditions for layer activation are only checked at join-points, and thus the

burden is on the programmer to ensure that their pointcuts match all message sends that may be relevant.

In the work presented here, any predicate expression can be continuously monitored, providing a flexible layer activation mechanism. In essence, any expression that returns a boolean in the source language can be associated with the activation state of a layer—the framework takes care to monitor all changes to the system that might change the outcome of that expression. We call these kinds of constraints *Layer Activators*.

Constraint Layers OCP languages are motivated by the observation that some issues in software development lend themselves to a declarative specification, while others are more directly expressed using imperative constructs. The goal is to provide both paradigms in a cleanly integrated fashion within an object-oriented language in a way that supports messages, encapsulation, inheritance, mutable objects, and object identity in constraints. In OCP, constraints can be defined, activated, and retracted at runtime. Using a meta-level protocol, constraints can be activated and retracted at arbitrary points during the execution. In addition, various OCP systems offer explicit scoping mechanisms for constraints.

Babelsberg[4] is a design for Object Constraint Programming languages, and this work is based on its JavaScript implementation, Babelsberg/JS[5]. Babelsberg defines three types of scopes for constraints: *always*, which defines and activates a constraint for the entire rest of the execution; *once*, which activates a constraint, solves it, and then immediately retracts it; and *assert–during*, which activates a constraint for the duration of a code block.

An issue with these scoping mechanisms is that they effectively allow the activation of constraints with global effects at any point in time. It is not possible to add restrictions on when constraints can be activated, making reasoning about a program difficult. Because constraints are, by their nature, multi-directional, any method may add a constraint involving its arguments and thus calling any method may affect the objects created by its caller.

To alleviate this problem and provide control over when constraints are active, we use COP to provide a convenient scoping mechanism to add, modify, or remove constraints using layers. We call such layers that adapt declarative constraints in addition to behavior *Constraint Layers*. Such a layer can add and remove constraints from the system, which are then continuously monitored and enforced throughout the execution.

***

The work presented here was done as a result of our experience in developing a browser-based clone of the game *Wii Play/Tanks!*, which provided opportunities to use both constraints and layers, but also exposed shortcomings of these language extensions in our use-case (Section 2). Since our target platform is the browser, our approach extends the Babelsberg/JS and ContextJS[11] implementations of OCP and COP in JavaScript (Sections 3 and 4), but we also discuss how our approach relates to other COP implementations (Section 5). While we have found our mechanisms to be useful in developing our game clone, we also report on open questions and future work in that direction (Section 6).

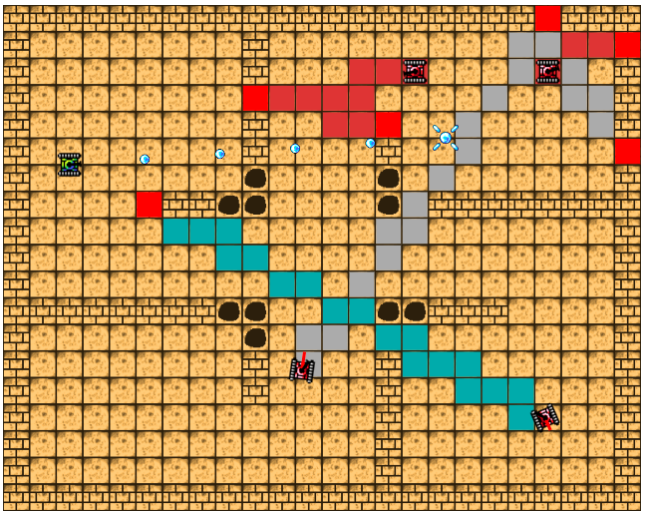

Figure 1. WePlayTanks with Babelsberg and ContextJS

2. Motivation and Goals

We set out to create a clone of the game *Wii Play/Tanks!*[1] that runs in the browser. The game is top-down, with the players controlling toy-tanks that shoot rubber bullets at each other. The tanks can move about the field and turn their turrets independently. The field contains obstacles such as walls and holes in the ground which the tanks cannot cross, but bullets can ricochet off walls and fly over holes. As an extension to the original, we added power-ups which give the tanks' bullets additional behavior such as flying faster or ricocheting more often.

The finished game is shown in Figure 1. From our experience with both COP and OCP, we realized there are multiple opportunities to use constraints and layer-based behavioral adaption. Power-ups change the behavior of bullets, and, during development, alternative drawing modes are desirable for introspection and editing. These can be conveniently encapsulated using layers. Additionally, there are UI elements that have constraints on them, such as the target following the mouse, or that a tank cannot drive into a wall or over a hole. We encapsulated these using constraints.

2.1 Desired Form of Layer Activation

Consider the use of layers for non-permanent power-ups, which are collected and activated in the game and should be deactivated after some time. Activating the layers imper-

[1] ©2007 by Nintendo®, `http://www.nintendo.com/sites/software_wiiplay.jsp`, accessed March 30, 2015

atively is straightforward: when the player drives the tank over the power-up, the corresponding layer becomes active. The case for deactivating the constraint, however, is not as clear.

To later deactivate the constraint, we could create a timeout which fires later to deactivate the layer [13]. This, however, makes the implementation to accumulate the times for multiple power-ups more difficult, as each timeout would have to check if the layer should indeed be deactivated or if more power-ups of the same type have extended the timeout. Alternatively, we could use a variable that remembers the last time the player found a power-up, and find a convenient place in the game loop to insert a check if the layer should be deactivated, with the drawback that we mingle different concerns into our game loop. Instead, we would like to re-use the existing game loop time and decide if a layer is active based on the time difference between the last collected power-up and the current game time. Intuitively, we might have a declarative definition of when the layer should be active, given a variable $lastCollectionTime$ which holds the last time the player collected a power-up:

$$lastCollectionTime + timeOut \leq gameTime$$

Given this definition, we can control both activation and deactivation of the layer. When the player drives over the power-up, we set $lastCollectionTime$ to $gameTime$, which, for a non-zero $timeOut$, implicitly activates the layer. As time progresses, the inequality will become false and implicitly, the layer is deactivated.

2.2 Desired Form of Constraint Scoping

In our game, consider that the turret of the tank should follow the mouse cursor controlled by the player. The constraint can be expressed in Babelsberg/JS as:

Listing 1.

```
1 always: {
2   player.turretDirection.equals(input.mouse.sub(player.position))
3 }
```

Constraints in Babelsberg/JS are ordinary JavaScript expressions that are introduced using the keywords `always` or `once`, the constraint being that the expression evaluates to true. Line 2 is the boolean expression that will be converted by the framework to be solved. This code is valid, although uncommon JavaScript, using `always:` as a code label. If this code is executed, the label is skipped and the expression simply run, no constraints will be created. In Babelsberg/JS, a translation mechanism interprets the expression while keeping track of the dependencies between encountered variables and fields[4]. Besides allowing for a more convenient syntax, the source code transformation creates a context object that contains all variables that are directly referenced by the function[5]. The framework requires this context object in order to interpret and transform the constraint expression. The transformed code for Listing 1 is:

Listing 2.

```
1 bbb.always(
2   {solver: new DBPlanner(),
3    ctx: { player: player, input: input }},
4   function() {
5     return player.turretDirection.equals(
6       input.mouse.sub(player.position)
7     );
8   }
9 );
```

As long as a constraint is active, the system keeps track of the objects that relate to it and, if the constraint becomes invalidated through modifications to those objects, triggers the solver to find a solution that satisfies them. This constraint will ensure that the turret direction will always be towards the mouse cursor. However, in the game we only want this to be the case as long as we are actually *playing*, not if the game is paused or we are editing the playing field. Yet, deactivating a constraint is currently only possible using meta-level access to constraints, and may be considered the equivalent of a `goto` in OCP. Instead, we need a scoping mechanism with which we can group, activate, and deactivate constraints.

Using layers in combination with our Activators, we can declare that play related constraints such as the one above should only be active if the current game state is playing, and have the activation mechanism take care of adding and removing them at appropriate times.

3. Layer Activators

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. We want to specify the relation between the behavior adaption and the condition explicitly. To do so, we introduce the concept of *Layer Activators*.

Activators activate a given layer as long as the associated constraint expression is fulfilled, as illustrated by Figure 2. In this figure, a change in the system state changes the result of predicate p to true. The system automatically activates the layer and execution continues in the system state that is augmented by the layer. When an execution step changes the result of the predicate, we deactivate the layer again. Note that the activation and deactivation is *immediate*. There is no time where any part of the program could observe a state where the condition is satisfied, but the layer is not active.

In order to implement this, we extend the existing Babelsberg library. Babelsberg is built to use multiple solvers, and let them cooperate to solve constraints. We can view the

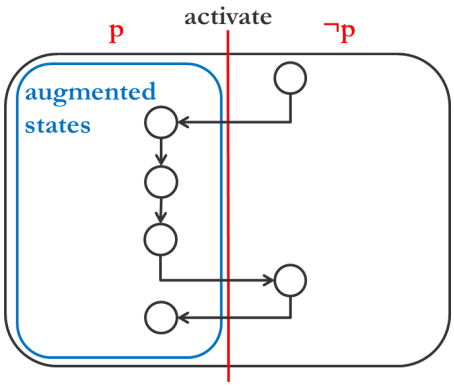

Figure 2. System state trajectory augmented by an activator constraint

condition to activate a layer as a constraint of the form —

$$booleanExpression? = isActive(Layer)$$

The questionmark means that the solver is not allowed to change the condition, and thus must change the activation state of the layer when the condition changes. Accordingly, we implement a special `ReactiveSolver`, which re-uses the Babelsberg framework for interpreting the condition expression, but has special knowledge to solve constraints involving the active state of a ContextJS layer. Whenever a variable that participates in the boolean expression changes, the framework asks our solver to `solve` the constraint. To do so, our custom solver checks whether the evaluation result of the condition coincides with the layer activation status and if not, activates or deactivates the layer as necessary.

Using this special solver embedded in Babelsberg/JS, a Layer Activator may look as follows:

```
1 always: { solver: new ReactiveSolver()
2   my.condition == aLayer.isActive()
3 }
```

This uses the standard Babelsberg/JS mechanism for constraint solving: the predicate my.condition == aLayer.isActive() is interpreted to determine its dependencies, and the objects that affect the outcome of the predicate are wrapped to monitor changes to them [5]. When the outcome of the predicate is false, the ReactiveSolver is called to change the participating objects to make it true—in this case, the solver knows that to do so means to adjust the activation state of aLayer to match the other side of the equality.

In practice, we found that syntactic sugar is useful for these kinds of constraints. The right-hand side of the constraint will always be an equation to the result of calling `isActive` on a layer. To make this clearer, we adopted a more convenient syntax. Using this syntax, we can use one of our power-up layers (Section 2.1) as follows:

Listing 3.

```
1  var powerUpLayer = new Layer().refineObject(tank, {
2    getBulletRicochets: function() {
3      return cop.proceed() + 1;
4    }
5  });
6  powerUpLayer.timeout = 6000;
7
8  activator(function() {
9    return timer.time > player.powerUpCollectionTime +
10                        powerUpLayer.timeout;
11 }).activates(powerUpLayer);
```

Lines 1–6 define a layer for a particular power-up. Lines 8–11 define a Layer Activator. It references the global timer and the player, and declares that when the predicate evaluates to true, the `powerUpLayer` should be active.

Since the Layer Activators are built around a shared constraint solving mechanism, the semantics for competing or contradictory activators arise out of the theory of constraint solving for contradictory or competing constraints. If two Activators contradict each other, solving will fail with an exception at the time the second Activator is defined—with the effect that the second Activator is not enabled.

4. Constraint Layers

Constraint Programming (CP) typically revolves around describing a global system state, which is why `always` and `once` constraints are often sufficient. In contrast, Object-oriented Programming (OOP) revolves modularization of system state and dynamic adaption. Babelsberg provides means to globally enable or disable constraints using the meta-level methods `enable` and `disable`, defined on the `Constraint`. As a limited scoping mechanism, constraints can be activated for the dynamic extent of a code block. This mechanism, however, is control flow-specific. Instead, we want to re-use the declarative form of activating layers for constraints.

As we integrate constraints and Object-oriented (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, analogous to partial classes and their behavior in behavioral layers, we introduce scoped constraints in *Constraint Layers*.

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 in the editor mode: as long as the game is in editor mode, the players cursor should always correspond to the `Tile` under the mouse. 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 cross-cuts all relevant modules to implement the editor mode including the constraint described above. Listing 4 shows the definition of this constraint:

Listing 4.

```
1 EditorLayer.addConstraint(
2   {solver: new DBPlanner()},
3   function() {
4     return cursor.tileIndex.equals(
5       map.positionToCoordinates(input.position)
6     );
7   }
8 );
```

Line 1 adds a constraint to the `EditorLayer` using a `addConstraint` method, which we have defined. This method takes a configuration object (Line 2), which in this case specifies that the predicate, if activated as a constraint, should be solved using the DeltaBlue planning solver. As second argument we pass the predicate function, which tests that the `tileIndex` field of the `cursor` is equal to the mouse position when converted to tile-based coordinates. When the EditorLayer is activated, this predicate is transformed into a constraint and solved for. - Using the `addConstraint` function, multiple constraints can be added to a layer. Thus, layers can group multiple constraints to atomically en-/disable multiple constraints related to the same context.

5. Related Work

We discuss work related to our layer activation mechanism. To our knowledge, there is no previous work that combines COP and Constraint Programming.

JCop [1] and *EventCJ* [8] separate the control of layer activation and the execution of context-dependent behavior using two language constructs, *event declarations* and *layer transition rules*. 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. EventCJ events rely on a subset of AspectJ[10] pointcuts. However, the pointcut has to be specified explicitly. In contrast, our Layer Activators allow to specify an arbitrary condition and, thereby, abstract from concrete point of execution to activate the layer. A benefit of EventCJ is that it provides instance-specific layer activation, while our work activates layers globally.

PyContext[14] introduces implicit layer activation, to address the issue that, similar to our observations, layers are often activated depending on a condition. If this condition changes frequently, the condition and activation may be scattered across the code. Each PyContext layer may define the `active` method and, if this method evaluates to true, the layer is active. Whenever a layered method is called, PyContext first determines which layers are active using those methods. In contrast, our work determines the composition when the condition changes. Depending on the frequency, one or the other implementation may incur more overhead.

ServalCJ[9] attempts to unify different 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 both dynamically scoped and reactive event-based activation mechanisms.

6. Discussion and Conclusion

We have presented a novel COP layer activation mechanism related to reactive and event-based layer activation techniques, and a novel use of layers as a scoping and control mechanism for constraints in OCP languages. These mechanisms resulted from our experience in creating an interactive game using both constraints and layers.

This work is practical, and open issues remain regarding semantics that were not relevant for our use-case. Addressing these issues will require further experience with our mechanisms.

First, it is currently not clear what happens when two conflicting Layer Activators refer to the same layer. The layer may active if one or more of the activator constraints evaluate to true, or only if all associated activator constraints evaluate to true. Alternatively, we may detect a conflict when trying to define an Activator for a layer that already has one. As a final option, we could ask the system to solve for all Activator expressions to be equal, when multiple expressions are defined.[2]

Second, how do Layer Activators and control flow-based activations of a layer interact? If a layer that is associated with an activator constraint is activated manually, we may raise an error or just fail silently if the activator expression is not true (the latter option being the practical expression of an assumed artificially small time between the imperative activation and the time when the Activator deactivates the layer). Alternatively, we may propagate the layer activation and prompt the framework to solve for the Activator expression to be true.[2]

Third, consider a Layer Activator that is part of a Constraint Layer, and both are active. What happens if the Constraint Layer is deactivated, and thus the Layer Activator as well. We see two options: 1) disabling the Activator deactivates the layer regardless of the current value of the Activator expression, or 2) disabling the Activator leaves its layer active.

Despite these open questions, our practical implementation of these mechanisms enabled us to construct a usable

[2] The semantics of the first two issues are clear when using the layer activation semantics of ContextJ[2]. In this implementation, multiple activations cause the behavioral variations to be applied multiple times. Thus, activating a layer twice through manual activation or multiple Layer Activators simply applies the behavior adaption twice.

game. We were able to use our mechanisms to make use of OCP constraints and COP layers in a way which improved the clarity and conciseness of our code, and believe that further work in this direction may benefit other application domains.

References

[1] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, and K. Kawauchi. Event-specific software composition in context-oriented programming. In *Software Composition*, pages 50–65. Springer, 2010.

[2] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. Contextj: Context-oriented programming with java. *Information and Media Technologies*, 6(2):399–419, 2011.

[3] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *Proceedings of the 2005 Symposium on Dynamic Languages*, pages 1–10. ACM, 2005.

[4] T. Felgentreff, A. Borning, and R. Hirschfeld. Specifying and Solving Constraints on Object Behavior. *Journal of Object Technology*, 13(4):1–38, 2014.

[5] T. Felgentreff, A. Borning, R. Hirschfeld, J. Lincke, Y. Ohshima, B. Freudenberg, and R. Krahn. Babelsberg/JS: A Browser-Based Implementation of an Object Constraint Language. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, pages 411–436. Springer, 2014.

[6] F. Geller, R. Hirschfeld, and G. Bracha. Pattern matching for an object-oriented and dynamically typed programming language. Technical Report 36, Universitätsverlag Potsdam, 2010.

[7] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3): 125–151, 2008.

[8] T. Kamina, T. Aotani, and H. 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*, pages 253–264. ACM, 2011.

[9] T. Kamina, T. Aotani, and H. Masuhara. Generalized layer activation mechanism through contexts and subscribers. In *Proceedings of the 14th International Conference on Modularity*, pages 14–28. ACM, 2015.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, pages 327–354. Springer, 2001.

[11] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Science of Computer Programming*, 76(12): 1194–1209, 2011.

[12] H. Ossher and P. Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM, 2000.

[13] M. Taeumel, T. Felgentreff, and R. Hirschfeld. Applying data-driven tool development to context-oriented languages. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, pages 1–6. ACM, 2014.

[14] M. Von Löwis, M. Denker, and O. Nierstrasz. Context-oriented programming: beyond layers. In *Proceedings of the 2007 Symposium on Dynamic Languages*, pages 143–156. ACM, 2007.