Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Behavioral Programming for Distributed Reactive Systems - Evaluation by an Analogy to Computer Game Design

Bachelorarbeit

im Studiengang Informatik

von

Caspar Prasuhn

Prüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Dr. Matthias Becker
Betreuer: Prof. Dr. Joel Greenyer

Hannover, 18.04.2015

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 18.04.2015

Caspar Prasuhn

Abstract

To find a way to overcome the complexity of distributed reactive software systems, I looked at "Behavioral Programming" (BP), which is a novel programming paradigm that claims to make the development of software easier in a natural and incremental way. My goal was to evaluate BP and find out its strengths and weaknesses, by taking a look at five criteria: (1) Incremental work flow, (2) changeability and extensibility, (3) modularity and readability, (4) performance and finally (5) difficulty of use. I did that by creating my own case study (the development of a traffic simulation with the Unity3D game engine and C-sharp as programming language) and then discussing the impressions that I gathered during its development. My results are the following: (1) BP allows for an incremental work flow, as long as the requirements are precise and complete. (2) BP makes software easily changeable and extensible. (3) BP forces modular and readable code. (4) BP has performance problems in larger systems but that can partly be solved with the concept of "nodes". (5) To understand BP and use it efficiently, one might need to invest some time to fully understand its principles and use them correctly.

Contents

Chapter 1

Introduction

1.1 Problem

Nowadays computer systems are all around us. These ubiquitous systems usually consist of many components, which themselves have several functions and interact with other components. When a system's behavior emerges from the interaction between several subsystems, we call it a distributed system. Such systems may possess increased complexity and the programming of such a system can become a difficult task.

A conventional way to deal with this in software engineering is the formulation of use cases, which the software is then build from. The software engineer would choose a use case and try to implement it. Usually object oriented programming (OOP) is used to divide the software in modules, so the software engineer would identify the necessary modules that are required for a certain use case and implement them incrementally one after another. Using that approach, a problem may arise: When a second use case has to be implemented, but that use case affects the functionality of the first use case, or vice versa. Then the software engineer can no longer focus solely on his second use case, but instead has to look at the modules of the first use case again. He then needs to figure out a solution, so that both use cases are implemented properly. Exactly this going back and forth between existing and new code disturbs the incremental work flow, causes complexity and thus might be time consuming and make room for errors. There is no pattern for the software engineer to follow in OOP to easily implement the conflicting use cases correctly, but instead he has to manually work it out and try to find a solution. Complex behavior is even defined as *several simple behaviors that conflict* by Jackson[9], which perfectly matches the situation described above. And especially in distributed systems, interwoven and conflicting behavior may exist in many parts of the system, so the software engineer constantly faces this problem.

1.2 Approach

A common approach in software engineering to deal with complexity is "divide and conquer". This means that you decompose your problem into several smaller ones, that are then more easily manageable, as stated by Peña et al.[11]. Jackson[9] however emphasizes that decomposition is a great tool to decrease complexity, but that the software engineer might miss interactions between decomposed parts of a software system by doing so. He calls this phenomenon "oversimplification". So if you take a look back at conventional software engineering, one could argue that use-cases and OOP provide some sort of decomposition of the big problem, but they both don't provide a nice way to model and take care of the joint behavior of decomposed parts.

That's why I want to take a look at and evaluate "Behavioral Programming", which is a programming paradigm that aims at encapsulating system behavior to allow for incremental development, one functionality at a time, to lower the need to go back to already implemented code. I want to find out if this programming paradigm decreases the complexity when implementing a distributed system, by investigating it's capabilities for (1) incremental development, (2) changeability and extensibility, (3) modularity and readability. In addition I want to take a look at (4) performance, as well as how difficult it is to for a novice to (5) understand, setup and use BP in a software project and what kind of problems my arise.

To find this out, my approach was to create use cases for a simple traffic simulation game and then implemented these using "Behavioral Programming" and the Unity3D engine. A traffic simulation game suits the field of application of distributed systems well, because on the one hand there are several objects (cars, pedestrians, traffic lights) that have their own behavior, which is then combined, forming the system's total behavior. On the other hand I took an element from computer games: The (human) player acts as a god with a top-down perspective and can interfere by manipulating the different objects and observe the system's reaction, to fully meet the requirements of a distributed and reactive system.

I chose this approach because it creates the case of having several use cases that affect each other. During the process of implementing these use cases I documented my findings and impressions to later evaluate the quality of the "Behavioral Programming" paradigm for reactive distributed systems by using the criteria (1) to (5) mentioned above.

1.3 Goal

The goal of this bachelor thesis is to evaluate the "Behavioral Programming" paradigm. Does it deliver what it promises? Does it allow for incremental

development and thus decrease complexity? Is it easier to make changes to or expand existing functionality, when using this paradigm? Does this paradigm lead to modular code that is easy to read?

1.4 Structure

This bachelor thesis has the following structure. In chapter 2 I explain the necessary foundations, especially how BP works and how it is integrated within Unity3D. In chapter 3 I present my case study (the traffic simulation) and showcase how i built it. Chapter 4 examines strengths and weaknesses of BP, by evaluating it on the basis of five criteria. In chapter 5 related work is discussed and compared to my work. Chapter 6 summarizes the most important findings and insights.

1.5 Results

The following results have been found:

1. BP allows for an incremental work flow, as long as the requirements are precise and complete.
2. BP makes software easily changeable and extensible.
3. BP forces modular and readable code.
4. BP has performance problems in larger systems but that can partly be solved with the concept of "nodes".
5. To understand BP and use it efficiently, one might need to invest some time to fully understand its principles and use them correctly.

Chapter 2

Foundations

2.1 Overview

Since my goal is to evaluate Behavioral Programming (BP), I will first describe what BP is, how it works and explain its components in detail. Also I will explain what Unity3D is and how it has been used in conjunction with BP to create the *traffic simulation game*.

2.2 Behavioral Programming

BP is a programming paradigm, a special approach and technique for software development with the goal to allow for incremental development in a natural way. In addition to conventional OOP components like *classes*, software designed after this paradigm also contains *Behavioral Threads* (BTs), each of which represents an independent scenario that the system should or shouldn't follow. For example, if you look at a game, each game rule would be programmed separately and independently in a BT, with little or no awareness of other BTs. A common execution mechanism, the *execution engine*, interlaces these behaviors at runtime, choosing events based on the demand and constraints of each BT, yielding an integrated system behavior. BP currently exists for several programming languages, including Java, Javascript, C++, Erlang and C sharp[8]. A more detailed explanation of BTs, events and the execution engine is given in the following sub chapters.

2.2.1 Behavioral Threads

A BT is a scenario that the system should or shouldn't follow. It can consist of usual program code but can also request, wait for or prevent **events**. Whenever one of these three **event** actions occurs, a *synchronization point* is reached. That means that the system executes all other BTs first,

until they hit a synchronization point themselves. When all BTs are at a synchronization point, the system chooses an Event that is requested and not prevented, and executes it. Now all BTs that requested the selected event, or waited for that event, will execute past their synchronization point until they reach their next synchronization point. All other BTs remain at their current state, until an Event is chosen that affects their synchronization point. BTs can consist of repeating or endless behavior (by putting their code in a "for" or "while" loop respectively), but they can also be finite.

2.2.2 Events

Events are the second core component of BP. In BP events can be requested, waited for or prevented. When all BTs reach a synchronization point, the execution engine looks for all events that are requested and at the same time not prevented by any BT. However, there might be more than one eligible event, but only one event can be executed at a time. In this case a mechanism must be chosen to select one event. This can either be done randomly, leading to non deterministic system behavior, or events may hold a priority value and the highest or lowest priority event is chosen. It is important to note that if you choose to go with the priority based approach, all existing events may never have the same priority value, or else you create non deterministic behavior again.

Whenever an event is chosen for execution, its run() method is executed. The run() method may contain program code that is then executed or may print console messages, but it doesn't have to. An event can always act as a dummy, having no executable code itself, but instead being there just to advance other BTs past their synchronization point.

Internally BP often has to compare events, for example to determine if a requested event is the same event as a prevented event. When you create a new event, you create a new class, thus giving it its own type, but it always inherits from the base "Event" class. You can add new variables to your own event if you need. Now when two events are compared, first its type is checked. If the type is the same, all variables for both events are checked and must contain the same value, in order to count as equal. This mechanism is great if you have several objects of the same type, for example cars in a traffic simulation, but you want the event to only affect one of them. You do not want to create a new event for each car that exists in your simulation. What you could do instead is add a variable "ID" to your event which connects the event with the car and now only events with the same type **and** the same "ID" count as the same. That way you can for example prevent an event for one car, but still allow that event for all other cars. From now on this technique will be called *parametrization of events*.

2.2.3 Execution Engine

Since software that is designed after the BP paradigm consists of several individual BTs, there needs to be a mechanism that interlaces all BTs and creates the system's total behavior. This is done in the so called *execution engine* (EE). The EE first looks at all BTs and finds all possible events. That means to go through all BTs, find all requested events and then check if these requested events are blocked by any other BT. All non-prevented and therefore eligible events are put in a *possible events* list. The next step is to choose one of the events of that list (this process has already been explained in the events section). Once an event is chosen, it will be executed. This will fire the event's run() method as well as advance all BTs that waited for or requested that event to their next synchronization point.

As I explained earlier, the EE takes all BTs and then starts its algorithm. The BTs are passed to the EE as a list. So if you want to add a new BT to your program, you have to add that BT to the list before the list gets handed over to the EE.

It is important to note that the computations done by the EE might be quite costly performance wise when you have a lot of BTs and events. In chapter 3 I will take a closer look at the performance of BP.

2.2.4 A Simple Example

To illustrate the how BP can be used, lets take a look at the two BTs shown below in the listing A.1 and walk through them step by step.

Listing 2.1: Pseudocode of two Behavioral Threads

```
public class BT_Car_Drive : BP.BThread {
   public void run()
   {
      while(true)
      {
         sync{ Requests = EV_Car_DriveRequest }
         Car.driveForward()
         sync{ Waits = EV_NextFrame }
      }
   }
}

public class BT_Car_KeepDistance : BP.BThread {
   public void run()
   {
      while(true)
      {
         if(Car.AnotherCarIsTooClose()){
            sync{
               Prevents = EV_Car_DriveRequest
               Waits = EV_NextFrame
            }
         }else{
            sync{ Waits = EV_NextFrame }
         }
      }
   }
}
```

The first BT (BT_Car_Drive) has his actions wrapped in a while(true) loop, which means that the actions inside will be repeated infinitely. The first action is a synchronization point in which it requests the event *EV_Car_DriveRequest*. If that requested event is chosen for execution by the execution engine, the BT will advance to its second action, which calls the driveForward() method on the Car class, which actually moves the car forward a short distance. This is **no** synchronization point, so no matter what, the BT will immediately advance to its third action, in which it waits for the event *EV_NextFrame*. Only when that event is executed, the BT will continue and start with its first action again.

The second BT (BT_Car_KeepDistance) also repeats infinitely. First, in its "if" statement, it calls the method AnotherCarIsTooClose() on the Car class to check if the minimum distance to another car is violated. If that's the case it prevents the *EV_Car_DriveRequest*, which affects the first BT, because there exactly this event is requested. The result is that the first BT

will not pass its first synchronization point and thus the car **will not move**, if the minimum distance to another car is violated. Also, it waits for the event *EV_NextFrame*, because you only have to prevent an event once per frame. If no minimum distance is violated, the BT just waits for the event *EV_NextFrame*, then begins from the start again. In that case, the first BT can successfully pass its first synchronization point and then move the car forward.

2.3 Unity3D and the integration of BP

My goal was to evaluate BP by creating a traffic simulation game. For that I needed an engine that supports 3D and allows for the integration of BP. Luckily, Unity3D supports programming in C sharp and BP was available in C sharp. Unity3D (http://unity3d.com) is a flexible 3D engine that can be used for all sorts of games and simulations and its basic version if free.

When integrating BP into Unity3D, I had some obstacles to overcome. Unity3D's architecture is a frame based one. Scripts in Unity3D have an update() method that is called by the engine every rendered frame. In that method you do all the calculations that need to be done (for example moving a car a tiny bit forward each frame in a traffic simulation). This gets repeated over and over every rendered frame. These rendered frames are also what you see on your monitor when executing your game or simulation. Now the questions was how to integrate BP's Execution Engine into this architecture. I chose to have one global "GameLogic" class, that runs the execution engine in its update() method, hence once every rendered frame. This way the EE and Unity3D are synchronized. Each frame the EE advances all BTs as far as possible. When all BTs are at a synchronization point and no more conditions to advance one of the BTs are met, the EE stops and "tells" Unity3D that its done with its calculation and that Unity3D can advance to its next frame. Precisely this transition from one frame to the next, is also the opportunity to add new BTs to the EE. For example you could check for user input and add new BTs depending on the input.

Figure 2.1 shows a sequence chart which illustrates how the execution engine is integrated in Unity3D.

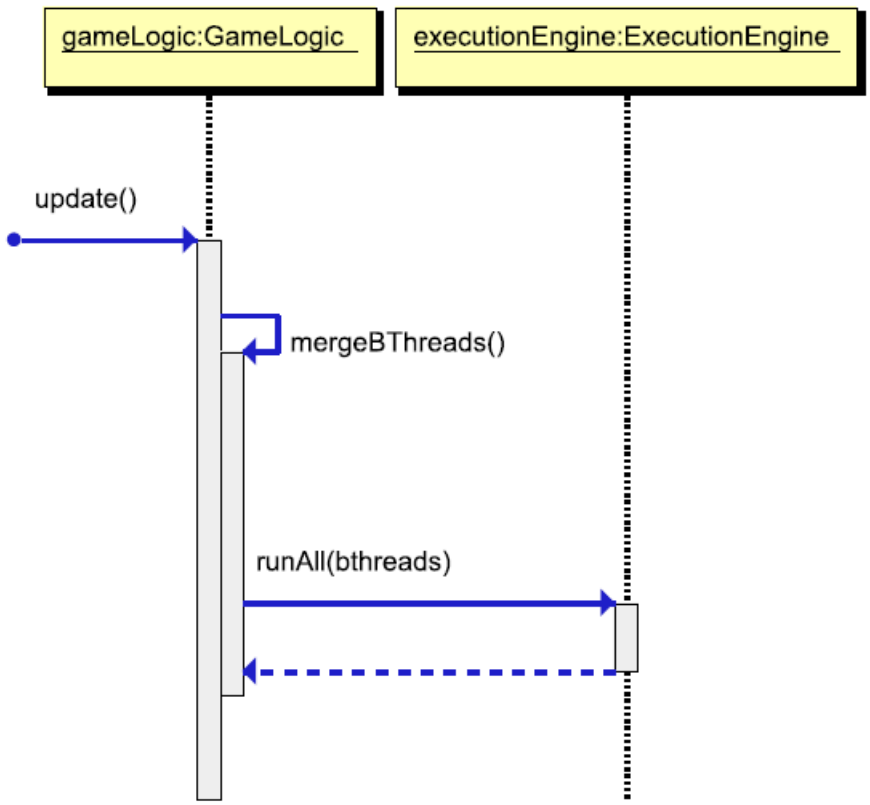

Figure 2.1: Implementation of the execution engine in Unity3D

As you can see, the first thing that happens is the call of the update() method, which is automatically done by the Unity3D engine. The next step is the mergeBThreads() method, which merges all new BTs that have been newly created, for example by user input, with all other BTs that are already existing. Now that all BTs are combined in one list, that list is passed to the execution engine in the runAll() method, and it can start its computation. By chosing this kind of implementation, one problem occured: When you had a BT that had repeating behavior and no BT prevented its requested events, it would run endlessly. The EE would execute that BT over and over again, so the EE would never finish and would never "tell" Unity that it's done with its execution. As a result Unity3D would never finish calculating that frame and the simulation would freeze. Because of this I created a "next frame" mechanism. That means that repeating

behavior may repeat, but only once per frame. Once it had its code executed, it must wait for the event *EV_NextFrame* to be allowed to be executed again. The *EV_NextFrame* event is always added once per frame in the mergeBThreads() method. This way the EE will not freeze on repeating behavior. If you take a look back at the code of the simple example A.1, you can see how I used the "next_frame" mechanism to make the car only move once per frame. The "next frame" mechanism I built for my implementation is similar to the "super-step" concept introduced by Harel et al [8].

Chapter 3

Case study: Traffic Simulation

In this chapter I will describe the traffic simulation that I developed. I will start with its basic functionality, then head towards the use cases I formulated and finally showcase the implementation including its Behavioral Threads.

3.1 Overview of functionality

The traffic simulation I created exists of a simple squarish road system. Cars automatically follow that road system in a circular fashion and they can be on either side of the road. Also there is a traffic light positioned at one of the roads which stops approaching cars if it shows red lights. The "player" can change the state of the traffic light, transitioning the red lights to green and vice versa. In addition there are pedestrians that automatically approach the traffic light, and if it shows a red light for the cars, the pedestrians will cross the road. The "player" can spawn additional pedestrians and additional cars via keyboard inputs. A screenshot of the application can be seen in figure 3.1 . Along the way of finding a concise example for my thesis I developed some further functionality (for example overtaking mechanisms, more BTs are shown in the appendix), but for the sake of keeping the following chapters more understandable, I will focus on a few core behaviors, that in my opinion highlight the application of BP well.

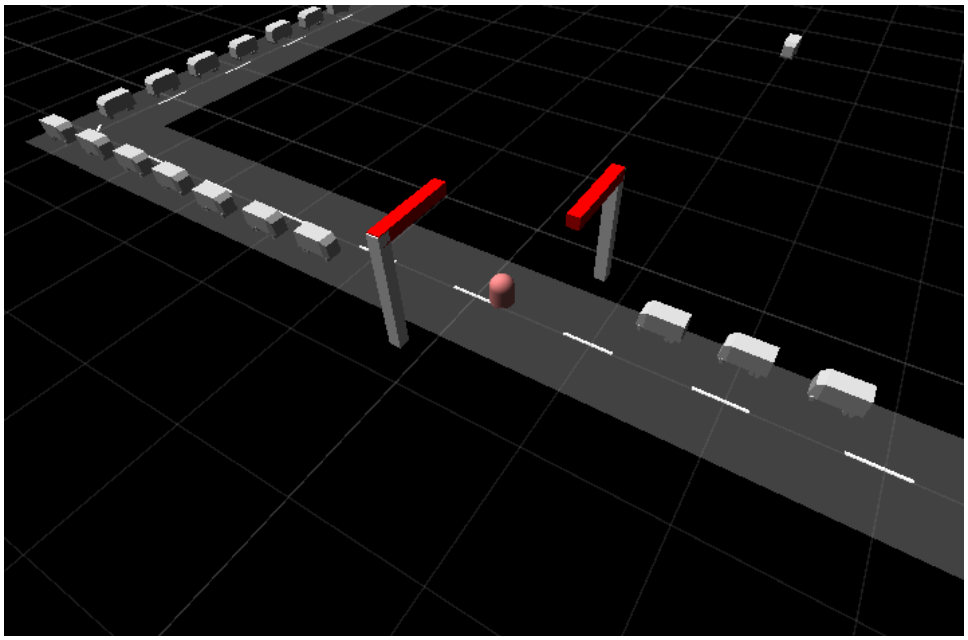

Figure 3.1: The traffic simulation with a traffic light, vehicles and a pedestrian crossing the street

3.2 Use Cases

Derived from the functionality described above, I formulated the following use cases:

Use Case 1: Cars can be spawned on the road system and they automatically follow it by "driving" forward with randomized speed.

Use Case 2: If a car detects another car in front of itself, it should always keep a certain distance instead of crashing into it.

Use Case 3: Pedestrians can be spawned via keyboard inputs which approach the traffic light and, if it shows red lights for the cars, cross the street.

Use Case 4: A traffic light is placed on one of the roads that stops approaching cars if its red lights are turned on, blocks pedestrians from crossing the street if its green lights are turned on and blocks cars and pedestrians if its orange lights are on. The player can toggle the traffic light's color from green to red via keyboard input and vice versa. Each transition will take 3 seconds, indicated by showing the orange light for 3 seconds.

3.3 Implementation using Behavioral Programming

I tried to incrementally build the software described in the use cases above, implementing one use case after another. I will not go into detail of how i setup the 3D environment of the simulation and how i connected the 3D objects with my code and classes in Unity3D, because the focus of this thesis is to evaluate the potential of BP, not to explain how Unity3D works. I also have to note that I have created a "GameLogic" class that keeps track of all dynamically created objects like cars and pedestrians, checks for player input, and runs the EE once each frame rendered by Unity, but I will not explain further details of how I did this and focus on the use cases instead.

The implementation of use case 1 and 2 has already been shown partially in the simple example A.1 in chapter 2, but was missing some extensions regarding *parametrization*, thus I will explain it in more detail once again. For **use case 1** I created the class "Car" which has the public method driveForward() that can be called from anywhere outside the class, moving the car forward a small distance. This class also automatically handles the pathing of the car to stay on the road at all times and drive on the correct side of the road. I then created a BT called "BT_Car_Drive" which in its first step requests the event "EV_Car_DriveRequest". This BT stores the ID of the corresponding car as an integer, to identify the correct car this BT belongs to (there might be several cars on the road and you want to be able to control the behavior of each car individually). This ID is also passed to the Event, for the same reasons. Whenever the EE selects that event for execution, the BT passes its synchronization point and driveForward() is called, which in fact moves the car forward in the simulation. After that, the BT waits for the event "EV_Next_Frame", because we only want the car to drive forward once per frame. The "next frame" concept has already been described in the foundations chapter and will not be explained in detail anymore. When the event "EV_Next_Frame" happens, the BT repeats itself, starting with a request for the event "EV_Car_DriveRequest". Because I do not want the car to stop, the steps of this behavior are nested in a "while(true)" loop.

For **use case 2** I implemented the method carInFrontOfThis() in the "Car" class, which checks if the minimum distance to any other car is on the same side of the road is violated. The BT created for this use case, called "BT_KeepDistance", then simply calls this function and if a violation occurs, simply prevents the event "EV_Car_DriveRequest", and if no violation occurs it just waits for the "next frame". It is important to note that this BT also stores the ID of a car, because you want to block the event "EV_Car_DriveRequest" only for the car where the violation happened. Because I never want this behavior to stop, it is also nested in a

"while(true)" loop.

For use case 3 I created the class "Pedestrian" which automatically lets pedestrians walk towards the traffic light and then stop right in front of it. That class also has the method crossTheStreet() which orders the pedestrian to cross the street, but only if it is already standing right beside the traffic light. I also created the BT "BT_CrossTrafficLight" which simply requests the event "EV_CrossTrafficLight". If that event is executed, crossTheStreet() is called on all pedestrians, making them cross the street, if they are close to the traffic light.

Use case 4 is an interesting one, because now the traffic light comes into action, which affects both cars and pedestrians. I created the class "TrafficLight" which stores the state of the traffic light (possible colors of the light shown to the cars: red, orange, green). That class also has the methods turnRed() and turnGreen() which cause the orange light to show for 3 seconds and then show the desired light. In addition, that class holds a list of all cars that are close to the traffic light and updates that list regularly. This list is needed to generate events to later prevent cars from driving forward when being close to the traffic light and the light being "red". Then I created the BT "BT_TrafficLightToggle" that handles the player input by waiting for the event "EV_TrafficLightToggleRequest" which is created when a certain keyboard key is pressed. Once that event is executed, turnRed() is called. Then, again, the BT waits for the toggle event and once that is triggered, turnGreen() is called. These steps are repeated infinitely. But here an additional BT is required: One that blocks cars and pedestrians depending on the traffic light's state. The BT "BT_TrafficLightBlockObjects" simply consists of 3 if statements that check for red, green or orange light and then prevents the events "DriveEventsOfAllCloseCars", "EV_CrossTrafficLight" or both respectively. This BT also repeats infinitely.

This completes the basic functionality of the traffic simulation. I later added more use cases to illustrate certain characteristics of BP, but these will described in the following chapter (Evaluation).

Chapter 4

Evaluation

In this chapter I will evaluate BP on the basis of the five criteria mentioned in the introduction. Each following subsection will evaluate one criterion.

4.1 Incrementality

My impression is that BP allows for a great incremental work flow. To support this theory I want to take a look at the implementation of **use case 4** of the previous chapter. When implementing that use case, I had to create the following:

- the new base class "TrafficLight" plus some methods for that class
- a new BT to toggle the traffic light on user input
- a new BT that stops cars and/or pedestrians

At no point I had to go back to a different base class other then the newly created "TrafficLight" class. The only thing that I had to know from previous implementations were the names of the events that the cars and pedestrians were requesting, so I could **prevent** these events.

It is important to note though that you can end up in situations where a previously implemented BT does not provide as much control as you need. One example where I faced this problem was when I wanted to make cars overtake other cars as illustrated in figure 4.1. This example is described in more detail in the following "Changeability and Extensibility" sub chapter, but the general problem was that I only wanted to prevent only the first step of the overtaking process, instead of the whole process. This way a car that started the overtaking process will always finish it and not stop in the middle of the process, possibly blocking the opposing side of the road. What I had to do then was to go back to an already implemented BT and change it according to my new needs. Also I had to make changes to the Car class, to make the new requirements work.

As a result, at first glance I would argue that situations may arise where the incremental work flow might be hindered. But looked at from a different perspective, I would also claim that in this case the incremental work flow was hindered because the requirements were imprecise and uncompleted. First I just thought of the overtaking process as one simple action, and only later I realized that there was more to it and that I needed to divide it in several different phases. Had I spent more time thinking about the requirements from the outset, I might have avoided the problem of lacking control later in the development process.

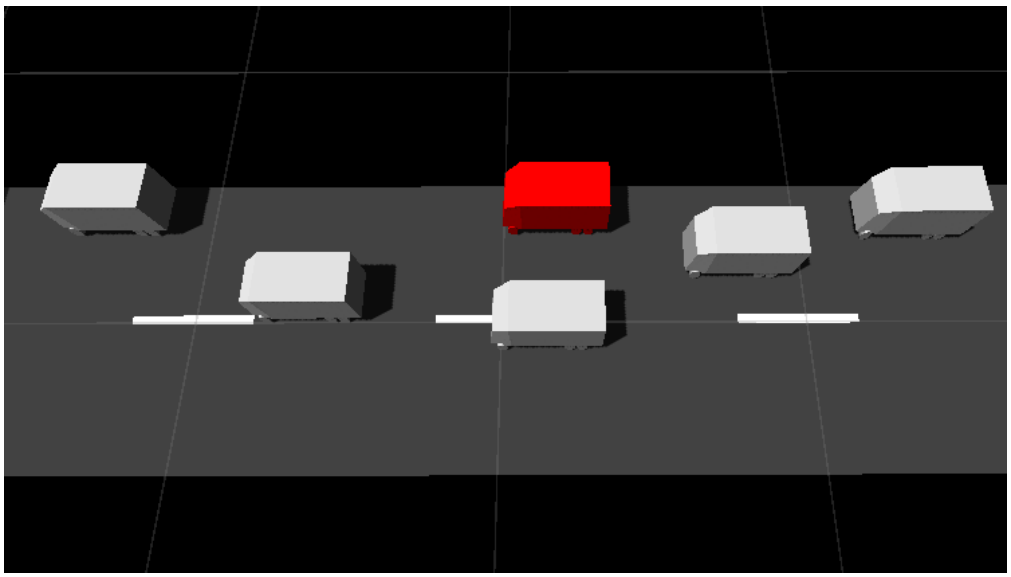

Figure 4.1: Vehicles overtaking a broken (red) vehicle, creating a dangerous situation by driving on the opposing side of the road

4.2 Changeability and Extensibility

First I want to talk about why changeability and extensibility is important in software engineering. On the one hand, in most cases, you want to make your software changeable and extensible so that standard maintenance procedures or additional customer demands can be met with as little effort as possible. In fact, Bohner et al.[1] state that the "change tolerance" of a software system is a factor that can decrease complexity and make development easier for the software engineers in the long run.

On the other hand, the phenomenon of "underspecification" must be considered, which means that in software engineering a requirements document is never totally complete and new requirements emerge in the process of building the software, as described by Harel[4]. So this means

that during software development, your software might have to change or be adjusted to meet the new requirements that came into existence. So if the software allows for easier changeability, the software engineer faces less complexity.

I think that BP makes changeability and extensibility relatively easy. On the one hand you can easily modify existing software by removing or interchanging BTs, without any need to look at a base class. When initializing and spawning a car, I could for example not create the BT "BT_KeepDistance" for that car, which leads to the car not caring for the minimum distance to other cars. This means I would have to change one line of code to change the behavior of that car, no further adjustments would have to be made.

On the other hand it's just as easy to extend existing software, just by adding new BTs. For example, I later added the ability for the player to click on a car to make it stop and turn its warning signs on, signaling to other cars that this car is blocking the road. I just implemented this by creating a new BT that blocks the clicked car's event "EV_Car_DriveRequest". However, this was a really simple extension. So I tried to create another more complex behavior. My idea was to make cars, that wait behind the stopped car because of the "keep distance" behavior, overtake that car instead of waiting endlessly. To achieve that behavior, I created a new BT "BT_Overtaking". To make this more complex behavior work, I also had to go back to the "Car" class and first add a new variable to track if a car got stopped by the player (a simple boolean is sufficient). Second, in the same class, I created the method carInfrontOfThisHasWarningSigns() which checks if there is a car blocking the road we are on. Third, the "Car" class needed a method overtake() that actually moves the overtaking car past the blocked car, basically in three steps: step out, drive forward and finally step back in. Now, with the needed functionality implemented to the base class, I could finish the BT: It first checks for a car blocking the road. If one is found it requests the new event "EV_Overtake" while preventing the event "EV_Car_DriveRequest", because we don't want the car to drive forward and crash into the car in front of it while overtaking. When the requested event "EV_Overtake" gets executed, overtake() is called and the process of overtaking begins.

With this new behavior, some simulation related problems did arise: For example, a car that overtakes a stopped car might crash into a car on the opposing side of the street. One could implement functionality that checks if there are cars coming on the opposing lane, and then block the event "EV_Overtake" in case that happens. Also the act of overtaking could be divided into several phases, each with its own event, so that you can control precisely which action of the overtaking process you want to request or prevent. However, this extension won't enrich this example (I implemented this advanced behavior and the according BTs can be seen in the appendix).

Instead I want to focus on what this example showed to me. First, I realized that adding new behavior doesn't always simply mean adding a new BT. As described above, I had to back to the "Car" class several times and add new functions and state variables. This was nothing that could have just been implemented in the BT itself, so the "Car" class had to be revisited, I saw no way to avoid that. But what makes BP so good is the *control you have over already implemented behavior*. Simply being able to prevent the event "EV_Car_DriveRequest" during the process of overtaking with just one line of code is really elegant. Also, for example, I later found overtaking, when being close to the traffic light, lead to weird traffic situations, so I decided that the "TrafficLight" class just prevents the event "EV_Overtake" for all cars that are close to it. The logic of the process of overtaking doesn't have to be touched at all, the simple prevention of the event "EV_Overtake" was sufficient.

To see this in a more abstract way, you can basically create any complex behavior and you just add a "marker" to it (the event it requests is the marker). And this marker event lets you prevent that complex behavior from anywhere with just one simple line of code.

4.3 Modularity and Readability

I put modularity and readability together in one section, because I think one great advantage of modularity is that it creates nice readability, and readability leads to easier understanding of software, especially if shown to people not familiar with the software. Because you usually create new BTs when adding new functionality to your software, you automatically end up with modular software. Also I find the modularity induced by the BTs quite feasible. Often the name of the BT alone gives insight in what it does (in my implementation, the names of the BTs are often close to the actual use case). Then, when looking into the BT itself, you can see what it does and in which order. Even if you don't have a clue how some behavior works in detail, you can control it (preventing the events it requests or waits for). Compared to software created conventionally with just OOP standards, you often have classes that have several functions and you sometimes don't know where to start when trying to understand what it does. BP on the other hand provides you with a place to start, by letting you look at the BTs, which summarize the features of the software quite well in my opinion.

Maybe it's appropriate to see BTs as a connector between complex low-level functionality and high-level descriptions of what the software does. In fact, the BTs I created never possessed complex logic themselves (the complexity never exceeded a few simple "if" statements), while the really complex calculations of the traffic simulation were placed in the base classes ("Car", "Pedestrian" and "TrafficLight").

4.4 Performance

I think one problem of BP lies in its capability to scale in larger systems. If you take a look at its execution engine, each iteration it does the following: (1) Go through all BTs, (2) for each requested events in these BTs, (3) go through all BTs to check if that event is prevented. If your system has n BTs and, for simplicity reasons, each BT always requests one event (it could theoretically request zero, but also an infinite amount of events), this would lead to a run-time of n^2. I benchmarked my simulation by periodically spawning more cars (each Car had 4 BTs at that point of time) and in fact, I could discover a quadratic increase in time required to calculate one frame, as seen in figure 4.2. Once I had more than 100 cars simultaneously driving down the roads, the frame rate dropped severely (less than 1 FPS).

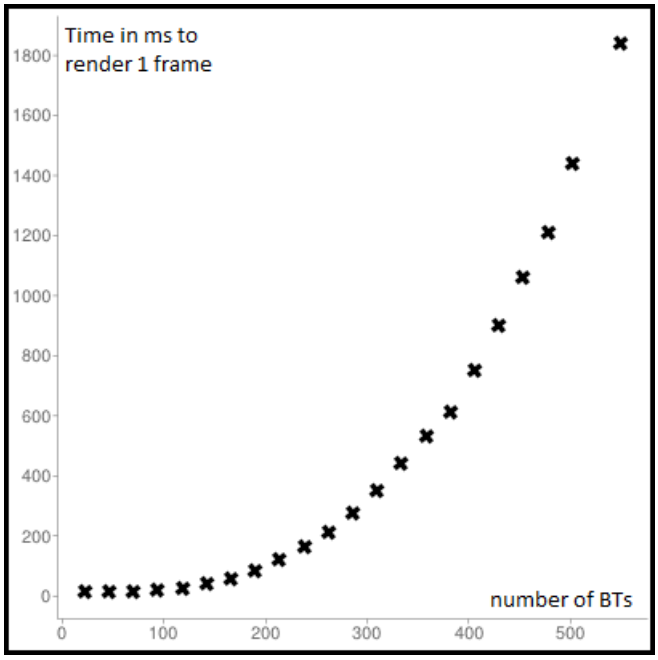

Figure 4.2: Time required to render one frame as a funtion of the number of Behavioral Threads

One possible solution to this problem is the creation of nodes, as proposed in Harel et al. [8]. This means that only behavior that needs to synchronize with each other is placed in the same node, decreasing the amount of iterations the "for loop" in (1) and (3) has to go through. Splitting your BTs in two equally sized groups and then running the EE individually for each node would lead to $(\frac{n}{2}*\frac{n}{2})+(\frac{n}{2}*\frac{n}{2}) = \frac{1}{2}n^2$, bisecting the amount of iterations necessary. However the question is, if you can always separate your behavior in different nodes. In some cases that might work, but maybe

sometimes that is not possible if they affect each other.

4.5 Difficulty and Learning Curve

I added this section because I wanted to share my impressions regarding the difficulty of learning and using BP. First of all to be clear, I was totally new to BP. During the work on my thesis I had roughly three months to understand and learn BP, and then integrate it in Unity3D to make my traffic simulation. At the beginning there was a lot to learn: How does the execution engine work, how often and when should it be run, when are events considered equal, when should super-steps happen, how to include user input, which event gets selected and executed first, how to connect BTs and basic classes, how can I make one car stop but not all of them, just to name some questions I stumbled upon in my "learning" phase. At first, this seemed a lot and almost overwhelming, and I also had the impression that BP creates a lot of overhead compared to conventional OOP: You don't only have to create your basic classes, but you also have to often create new classes for events and BTs when you implement something new. It was only after a few weeks of continuous engagement with the matter, that I learned to appreciate its power and see its advantages rather then seeing the negative aspects a novice might stumble upon. My conclusion would be that first, one needs to invest some time to get comfortable with BP and second, the elegance, control and modularity you gain when using BP also has its price by creating some overhead in the work flow.

In a recent case study by Harel et al.[5] it has also been highlighted that in order to make BP work in a large real-world system, some extensions had to be made. Among other things they mentioned parameterized events and the dynamic creation of threads, which are two features that I also used in my own case study. On the one hand this shows that BP is very flexible, and thus can be adjusted to meet your project's needs. On the other hand this might indicate that maybe BP could benefit from further improvements to make it more easily applicable to real-world systems and to be more accessible for novices.

Chapter 5

Related Work

An early attempt to decrease complexity in distributed reactive systems using scenarios of behavior is described in Harel et al.[6]. Here the idea is to not think in terms of individual objects and their states, but rather in terms of a typical scenario that could happen within a system. This is similar to the Behavioral Threads of BP, where one BT also describes one certain behavior of the system. They claim that this approach makes software development much more natural. Also, they introduce the idea of scenarios that *must*, *may* or *may not* happen, which has parallels to the concept of BP with its *requesting*, *waiting* and *preventing* of events. When the scenario-based behaviors were defined, they used a tool called the Play-Engine to *play in* these scenarios using live sequence charts and later *play them out*.

Another attempt using live sequence charts, this time in conjunction with actual behavioral threads, implemented in Java (called BPJ), is described in Harel et al.[7]. Here the concepts of requesting, waiting and preventing events are fully developed. Also, a coordination mechanism has been designed which synchronizes and interlaces the behavioral threads, yielding composite, integrated system behavior, which is similar to the C-sharp implementation that I used for my work.

Behavioral Programming as a novel and language independent programming paradigm, is nicely described and explained in Harel et al.[8]. Here, its functionality is shown in great detail. They also demonstrate its capabilities by explaining the incremental development of a computer game (the game "Tic Tac Toe"), much like I did in my work. They conclude that more research and projects using BP must be done to further evaluate its strengths and weaknesses. In addition, they propose that behavioral programming might be well suited for feature-oriented software, where customization is important.

This leads to the work of Kindler et al.[10], which takes a look at aspect-oriented programming, which aims at extending basic software with so called aspects. These aspects are programmed individually first and

later interactions and crosscutting behavior is modeled using *automata*. They also use events to synchronize different aspects, which is similar to the "synchronization point" mechanism of BP. One core benefit of aspect oriented is the flexibility of the developed software: Adding and removing parts (aspects) of the software is made easy and is possible without touching other existing parts of software. This benefit can also be found in BP, which also makes possible flexible software, by simply allowing the removal and addition of BTs, without having to adjust other parts of the software. One difference to mention though is that aspect-oriented programming focuses more on the implementation of individual classes, whereas BP focuses more on implementing scenarios of behavior. For a nice overview and further reading of aspect oriented concepts and technologies see Brichau et al. [2] and Chitchyan et al. [3].

Chapter 6

Conclusion

6.1 Summary

In this thesis I explained how BP functions and then used it to develop a traffic simulation, in which cars and pedestrians act autonomously, while a human "player" can interfere and manipulate them. Afterwards I evaluated BP, taking the impressions gathered during the development, using five criteria: (1) Incremental work flow, (2) changeability and extensibility, (3) modularity and readability, (4) performance and finally (5) difficulty of use. The results in a short are:

1. BP allows for an incremental work flow, as long as the requirements are precise and complete.
2. BP makes software easily changeable and extensible.
3. BP forces modular and readable code.
4. BP has performance problems in larger systems but that can partly be solved with the concept of "nodes".
5. To understand BP and use it efficiently, one might need to invest some time to fully understand its principles and use them correctly.

6.2 Outlook

In my opinion two problems regarding BP aren't fully solved and should be further investigated: (1) Can the performance problem be solved in really big and computationally intensive software systems? The concept of "nodes" is proposed as a solution, but does this help enough and in every software system? And (2), I personally felt that BP was not perfectly "ready to use". There were some problems I had to find solutions myself, and this didn't always feel easy. So my questions would be if there is a way to improve the

framework, so that some common problems are already solved. This would especially improve the accessibility for novice users in my opinion.

Appendix A

Ein Anhang

Below you can see the source code of all my Behavioral Threads created during the development of my traffic simulation.

A.0.1 Behavioral Threads

Listing A.1: Behavioral Threads source code

```
public class BT_Car_Drive : BP.BThread {

 int carID;

 public BT_Car_Drive(int car_id){
  carID = car_id;
 }

 public override IEnumerable<BP.RWB> run()
 {
  while(true)
  {
   Car car = GameLogic.cars.Find (x => x.carID==this.carID);
   yield return new BP.RWB {
    Requests = new List<BP.Events.Event>() {
     new EV_Car_DriveRequest(this.carID)}};
   car.driveForward();
   yield return new BP.RWB {
    Waits = new List<BP.Events.Event>() { new EV_NextFrame()}};
  }
 }
}

public class BT_KeepDistance : BP.BThread {

 int carID;

 public BT_KeepDistance(int car_id){
  carID = car_id;
 }
```

```
31
32  public override IEnumerable<BP.RWB> run()
33  {
34   while(true)
35   {
36    Car car = GameLogic.cars.Find (x => x.carID==carID);
37    if(car!= null && car.carInfrontOfThis()){
38     yield return new BP.RWB {
39      Waits = new List<BP.Events.Event>() { new EV_NextFrame()},
40      Prevents = new List<BP.Events.Event>() {
41       new EV_Car_DriveRequest(carID)}};
42    }else{
43     yield return new BP.RWB {
44      Waits = new List<BP.Events.Event>() { new EV_NextFrame()}};
45    }
46   }
47  }
48 }
49
50 public class BT_Overtaking : BP.BThread {
51
52  int carID;
53
54  public BT_Overtaking(int car_id){
55   carID = car_id;
56  }
57
58  public override IEnumerable<BP.RWB> run()
59  {
60   while(true){
61    Car car = GameLogic.cars.Find (x => x.carID==carID);
62    if((car.carInfrontOfThisHasWarningSigns() ||
63    car.ueberholPhase!=0) &&
64    !car.inactive){
65     if(car.ueberholPhase==1) yield return new BP.RWB {
66      Requests = new List<BP.Events.Event>() {
67       new EV_StepOut(carID)},
68      Prevents = new List<BP.Events.Event>() {
69       new EV_Car_DriveRequest(carID)}};
70     if(car.ueberholPhase!=1) yield return new BP.RWB {
71      Requests = new List<BP.Events.Event>() {
72       new EV_Overtake(carID)},
73      Prevents = new List<BP.Events.Event>() {
74       new EV_Car_DriveRequest(carID)}};
75     car.overtake();
76     yield return new BP.RWB {
77      Waits = new List<BP.Events.Event>() {
78       new EV_NextFrame()},
79      Prevents = new List<BP.Events.Event>() {
80       new EV_Car_DriveRequest(carID)}};
81    }else{
82     yield return new BP.RWB {
83      Waits = new List<BP.Events.Event>() {
84       new EV_NextFrame()}};
```

```
85      }
86     }
87    }
88   }
89
90   public class BT_Overtaking_SmartWarning : BP.BThread {
91
92    int carID;
93    float safetyDistance=2200f;
94
95    public BT_Overtaking_SmartWarning(int car_id){
96     carID = car_id;
97    }
98
99    public override IEnumerable<BP.RWB> run()
100   {
101    while(true){
102     Car car = GameLogic.cars.Find (x => x.carID==carID);
103     bool uberholenIsDangerous=false;
104     foreach (Car other in GameLogic.cars){
105      if(car.currentSide!=other.currentSide &&
106      car.carIsInfront(other) &&
107       Vector3.SqrMagnitude(car.transform.position-
108       other.transform.position)<safetyDistance){
109       uberholenIsDangerous=true;
110      }
111     }
112     if(uberholenIsDangerous) {
113      yield return new BP.RWB {
114       Waits = new List<BP.Events.Event>() {
115        new EV_NextFrame()},
116       Prevents = new List<BP.Events.Event>() {
117        new EV_StepOut(carID)}};
118     }else{
119      yield return new BP.RWB {
120       Waits = new List<BP.Events.Event>() {
121        new EV_NextFrame()}};
122     }
123    }
124   }
125   }
126
127   public class BT_TrafficLightToggle : BP.BThread {
128
129
130    public BT_TrafficLightToggle(){
131
132    }
133
134    public override IEnumerable<BP.RWB> run()
135    {
136     while(true)
137     {
138      yield return new BP.RWB {
```

```
139     Waits = new List<BP.Events.Event>() {
140       new EV_TrafficLightToggleRequest()} };
141    TrafficLight.turnRed();
142    yield return new BP.RWB {
143     Waits = new List<BP.Events.Event>() {
144      new EV_TrafficLightToggleRequest()}};
145    TrafficLight.turnGreen();
146   }
147  }
148 }
149
150 public class BT_TrafficLightKeyboardDetection : BP.BThread {
151
152  public BT_TrafficLightKeyboardDetection(){
153
154  }
155
156  public override IEnumerable<BP.RWB> run()
157  {
158   yield return new BP.RWB {
159    Requests = new List<BP.Events.Event>() {
160      new EV_TrafficLightToggleRequest()}};
161   yield return new BP.RWB { };
162  }
163 }
164
165 public class BT_TrafficLightBlockObjects : BP.BThread {
166
167
168  public BT_TrafficLightBlockObjects(){
169  }
170
171  public override IEnumerable<BP.RWB> run()
172  {
173   while(true)
174
175   {
176    if(TrafficLight.color==TrafficLight.red){
177     yield return new BP.RWB {
178      Waits = new List<BP.Events.Event>() {
179       new EV_NextFrame()},
180      Prevents = TrafficLight.listofclosecars };
181    }
182    if(TrafficLight.color==TrafficLight.green){
183     yield return new BP.RWB {
184      Waits = new List<BP.Events.Event>() {
185       new EV_NextFrame()},
186      Prevents = new List<BP.Events.Event>() {
187       new EV_CrossTrafficLight()} };
188    }
189    if(TrafficLight.color==TrafficLight.orange){
190     List<BP.Events.Event> list =
191      new List<BP.Events.Event>() {
192       new EV_CrossTrafficLight()};
```

```
193     list.AddRange(TrafficLight.listofclosecars);
194     yield return new BP.RWB {
195      Waits = new List<BP.Events.Event>() {
196       new EV_NextFrame()},
197      Prevents = list};
198    }
199   }
200  }
201 }
202 
203 public class BT_Car_WarningSigns : BP.BThread {
204 
205  int carID;
206 
207  public BT_Car_WarningSigns(int car_id){
208   carID = car_id;
209  }
210 
211  public override IEnumerable<BP.RWB> run()
212  {
213   while(true)
214   {
215    yield return new BP.RWB {
216     Waits = new List<BP.Events.Event>() {
217       new EV_Car_WarningSignToggle(carID)}};
218    yield return new BP.RWB {
219     Waits = new List<BP.Events.Event>() {
220       new EV_Car_WarningSignToggle(carID)},
221     Prevents = new List<BP.Events.Event>() {
222      new EV_Car_DriveRequest(carID)}};
223   }
224  }
225 }
226 
227 public class BT_Car_WarningSignsOff : BP.BThread {
228 
229  int carID;
230 
231  public BT_Car_WarningSignsOff(int car_id){
232   carID = car_id;
233  }
234 
235  public override IEnumerable<BP.RWB> run()
236  {
237   yield return new BP.RWB {
238    Requests = new List<BP.Events.Event>() {
239     new EV_Car_WarningSignToggle(carID)}};
240   yield return new BP.RWB { };
241  }
242 }
243 
244 public class BT_CrossTrafficLight : BP.BThread {
245 
246  public BT_CrossTrafficLight(){
```

```
247
248  }
249
250  public override IEnumerable<BP.RWB> run()
251  {
252   while(true)
253   {
254    yield return new BP.RWB {
255     Requests = new List<BP.Events.Event>() {
256      new EV_CrossTrafficLight()}};
257    foreach(Pedestrian p in GameLogic.pedestrians){
258     p.crossTheStreet();
259    }
260    yield return new BP.RWB {
261     Waits = new List<BP.Events.Event>() {
262      new EV_NextFrame()}};
263   }
264  }
265 }
266
267 public class BT_NextFrame : BP.BThread {
268
269
270  public BT_NextFrame(){
271  }
272
273  public override IEnumerable<BP.RWB> run()
274  {
275   yield return new BP.RWB {
276    Requests = new List<BP.Events.Event>() {
277     new EV_NextFrame()}};
278   yield return new BP.RWB { };
279  }
280 }
```

Bibliography

[1] S. Bohner, R. Ravichandar, and A. Milluzzi. Accommodating adaptive systems complexity with change tolerance. In M. Hinchey and L. Coyle, editors, *Conquering Complexity*, pages 49–72. Springer-Verlag London, 2012.

[2] J. Brichau and M. Haupt. Survey of aspect-oriented languages and execution models. Technical Report AOSD-Europe-VUB-01, 2005.

[3] R. Chitchyan, A. Rachid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, A. Jackson, and S. Clarke. Survey of aspect-oriented analysis and design approaches. Technical Report AOSD-Europe-ULANC-9, 2005.

[4] D. Harel. Can programming be liberated, period? *IEEE Computer*, 41(1), 2008.

[5] D. Harel and G. Katz. Scaling-up behavioral programming: Steps from basic principles to application architectures. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 95–108, New York, NY, USA, 2014. ACM.

[6] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[7] D. Harel, A. Marron, and G. Weiss. Programming coordinated behavior in java. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 250–274, 2010.

[8] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Communications of the ACM*, 55(7):175–198, 2012.

[9] M. Jackson. Simplicity and complexity in programs and systems. In M. Hinchey and L. Coyle, editors, *Conquering Complexity*, pages 49–72. Springer-Verlag London, 2012.

[10] E. Kindler and D. Schmelter. Aspect-oriented modelling from a different angle: Modelling domains with aspects. In *Proceedings of the 2008 AOSD Workshop on Aspect-oriented Modeling*, AOM '08, pages 7–12, New York, NY, USA, 2008. ACM.

[11] J. Peña, R. Levy, M. Hinchey, and A. R. Cortés. Dealing with complexity in agent-oriented software engineering: The importance of interactions. In *Conquering Complexity*, pages 191–214. 2012.