Creating, Tailoring, and Distributing Program Animations

Supporting the Production Process of Interactive Learning Content

Teemu Sirkiä
Creating, Tailoring, and Distributing Program Animations

Supporting the Production Process of Interactive Learning Content

Teemu Sirkiä

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 of the school on 22 September 2017 at 12 noon.

Aalto University
School of Science
Department of Computer Science
Learning + Technology Research Group
Supervising professor
Prof. Lauri Malmi, Aalto University, Finland

Thesis advisor
D.Sc. (Tech) Juha Sorva, Aalto University, Finland

Preliminary examiners
Prof. Thomas L. Naps, University of Wisconsin Oshkosh, USA
Prof. J. Ángel Velázquez-Ilturbide, Universidad Rey Juan Carlos, Spain

Opponent
Adjunct Professor Mikko-Jussi Laakso, University of Turku, Finland
Abstract

Modern web technologies have enabled new ways of producing interactive learning content. This content can be embedded into course materials on the web. Still the process to produce and distribute online activities is challenging for instructors. In this thesis, we introduce three new systems to support the process of producing learning content, distributing it, and letting instructors customize the content to their particular teaching goals and contexts. More specifically, we explore the topic in the domain of computer science education. We use program animations as a central theme. In addition to describing the technical aspects, we report preliminary results from CS1 courses which have used these systems to provide interactive content for learners.

The first part of the thesis presents the Jsee library for creating program animations which gives a visual representation of a notional machine and shows how the program state changes when a program is executed step-by-step. Animations are aimed at novice programmers who are learning the execution model. The library supports expression-level visualization, and because of its extensibility, it can support programming language specific visualizations.

In addition to the technical discussion, we present preliminary results how learners in two different CS1 courses used the animations made with Jsee. We use automatically collected log files as a primary data source to analyze the usage. Although learners did not receive any points, over 80% of novices voluntary viewed animations throughout the CS1 course which tightly integrated over 50 animations with other course materials.

The second part introduces Acos server for hosting and distributing reusable online learning content. We describe a new way to reduce the workload of developers by separating the content and the required communication between the content and learning management systems (LMS). Acos acts as a proxy which provides a uniform interface for the content but can use different interoperability protocols to communicate with the LMSs. In this way, the same content can be used with multiple LMSs, and the developer does not have to know which interoperability protocol is in use between Acos and the LMS. Acos has been successfully used to provide interactive content for CS1 courses, and the initial feedback from developers is promising.

The third part presents the Kelmu toolkit to tailor existing interactive animations. The animations are typically generic and often automatically produced. By using Kelmu, instructors can add, for example, textual explanations on top of the original animation to emphasize the integral parts of the animation and give explanations which are important in the instructor’s pedagogical context. This kind of signaling is supported by multimedia learning principles to enhance learning. Log data and eye-tracking traces give preliminary results that learners read annotations and pay more

Keywords program animation, program visualization, expression-level visualization, tailoring, distributing, online learning activities
Tekijä
Teemu Sirkiä

Väitöskirjan nimi
Ohjelma-animaatioiden luominen, mukauttaminen ja jakelu

Julkaisija
Perustieteiden korkeakoulu

Yksikkö
Tietotekniikan laitos

Sarja
Aalto University publication series DOCTORAL DISSERTATIONS 142/2017

Tutkimusala
Ohjelmistotekniikka

Käsikirjoituksen pvm
11.04.2017

Väitöspäivä
22.09.2017

Julkaisuluvan myöntämispäivä
12.06.2017

Kielu
Englanti

Monografia
Artikkeliväitöskirja

Esseeväitöskirja

Tiivistelmä
Modernit web-teknilöpöttävät ohjelma-animaatioiden luominen, mukauttaminen ja jakelu

Tässä väitöskirjassa esitellään kolme uutta järjestelmää, jotka ovat rakenteellisesti ja fuusioina Osiin 12


It has been a long but very interesting journey to write this thesis. I started in May 2009 as a research assistant in the LeTech research group (or its predecessor at that time). I completed my Master's thesis in June 2012 and became a doctoral student in December 2012. During all these years, I have been involved in teaching our CS1 courses as a teaching assistant or head assistant. Much of the work in this thesis has been done to improve the course materials and the learning experience.

There is a large number persons who I would like to thank. First of all, I am grateful for Professor Lauri Malmi for letting me be a member of the LeTech research group, giving me an opportunity to doctoral studies, and supervising this thesis. I want to thank my advisor D. Sc. Juha Sorva with whom I have worked together all these years. Lauri and Juha, your guidance, ideas, and suggestions have been invaluable for completing this thesis.

I am thankful that Professors Thomas L. Naps and J. Ángel Velázques-Iturbide had the opportunity to be the pre-examiners and had time to read this thesis, and provide their valuable comments and suggestions how to improve it. I am also grateful for having Adjunct Professor Mikko-Jussi Laakso as my opponent.

I want to thank all my colleagues, especially those who sat in the legendary room B207 more or less officially. Lassi, Teemu, Aleksi, and Otto, thank you for the support, help, and feedback you have provided. Kerttu Pollari-Malmi has always let me try new innovations in her course and collect data for research. It was also Kerttu who originally hired me at Aalto University (or the former Helsinki University of Technology) to be her course assistant before I started as a research assistant. Adjunct Professor Ari Korhonen guided me during Lauri’s sabbatical and Ari has also been one of the senior members of the group. Ville, Juha, Lasse, and
Preface

Tapio, you have shown as senior doctoral students a path to follow and have also given guidance. It has been a privilege to work with all of you.

I am grateful for all my co-authors. Participating the ITiCSE working group in Uppsala was a very interesting experience. Writing papers with Juha and Lassi has been fruitful and productive. I want to thank Professor Peter Brusilovsky for supporting this thesis and the co-operation with your students (especially Julio and Roya). The co-operation opened totally new possibilities and is an integral part of this thesis in the form of the Acos server.

Last but not least, I want to thank my family and relatives for supporting me during all these years. Our first PC arrived when I was eight years old. That took my programming hobby to a completely new level after working with the C64. That and the later computers were certainly a worthwhile investment!

Espoo, July 31, 2017,

Teemu Sirkiä
Preface 1

Contents 3

List of Publications 7

Author’s Contribution 9

Terms 11

1. Introduction 13

1.1 The Web as a Platform for Learning Resources 13
1.2 Program Animation as an Aid to Learn Programming 14
1.3 Distributing Reusable Online Activities 16
1.4 Tailoring Automatically Generated Animations 17
1.5 Supporting the Production Process 18
1.6 Research Questions 19

1.6.1 Problems in Program Animation 20
1.6.2 Problems in Distributing and Creating Reusable Online Activities 21
1.6.3 The Need for Tailoring Animations 22
1.7 Relationship with the Publications 23
1.8 Structure of the Thesis 23

I Creating Program Animations 25

2. Related Work 27

2.1 The Field of Software Visualization 27
2.1.1 Software Visualization 27
2.1.2 Program Visualization 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>Program Animation</td>
<td>30</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Why to Use Program Animation?</td>
<td>30</td>
</tr>
<tr>
<td>2.3</td>
<td>Program Visualization Tools</td>
<td>36</td>
</tr>
<tr>
<td>2.3.1</td>
<td>The Jeliot Family</td>
<td>38</td>
</tr>
<tr>
<td>2.3.2</td>
<td>UUhistle</td>
<td>40</td>
</tr>
<tr>
<td>2.3.3</td>
<td>ViLLE</td>
<td>42</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Online Python Tutor</td>
<td>43</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Summary</td>
<td>44</td>
</tr>
<tr>
<td>3.</td>
<td>The Need and Aims for the Library</td>
<td>45</td>
</tr>
<tr>
<td>3.1</td>
<td>Why a Library Instead of an Application?</td>
<td>45</td>
</tr>
<tr>
<td>3.2</td>
<td>Multiple Programming Languages</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>Expression-Level Visualization</td>
<td>47</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Granularity of Steps in Animation</td>
<td>47</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Examples of Expression-Level Visualization</td>
<td>48</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Teaching Expression Evaluation in Textbooks</td>
<td>49</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Teaching Expression Evaluation with Online Tutors</td>
<td>51</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Technical Challenges in Visualizing Expression-Level Evaluation</td>
<td>52</td>
</tr>
<tr>
<td>3.4</td>
<td>Embeddable Content for Web Pages</td>
<td>53</td>
</tr>
<tr>
<td>3.5</td>
<td>Features the Library Should Contain</td>
<td>54</td>
</tr>
<tr>
<td>4.</td>
<td>Jsvee Library</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Design Principles</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Supported Programming Concepts</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>Technical Implementation</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>Producing Visualizations with Jsvee</td>
<td>62</td>
</tr>
<tr>
<td>4.5</td>
<td>Empirical Evaluation</td>
<td>65</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Using Jsvee in CS1 courses</td>
<td>65</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Visualizations as Part of Online Materials</td>
<td>66</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Visualizations as Voluntary Practice Content</td>
<td>68</td>
</tr>
<tr>
<td>5.</td>
<td>Summary</td>
<td>71</td>
</tr>
</tbody>
</table>

II Distributing Online Learning Activities | 73 |

6. Problems in Distributing Activities | 75 |
| 6.1 | Interoperability and Reusability | 75 |
| 6.2 | Extensibility | 76 |
Contents

10.3 Using Kelmu ........................................ 105
10.4 Technical Implementation .......................... 106
10.5 Results .............................................. 108
  10.5.1 Using Kelmu Increases Time-On-Task .......... 108
  10.5.2 Novices Read Annotations ...................... 109
  10.5.3 Learning Effects ............................... 109
10.6 Kelmu with Other Animators .......................... 111

11. Summary ............................................. 113

IV Discussion ........................................... 115

12. Contributions ......................................... 117
  12.1 Developers' Point of View ......................... 118
    12.1.1 Creating Program Animations .................. 118
    12.1.2 Using Acos to Create Reusable Content ....... 119
  12.2 Instructors' Point of View ......................... 120
    12.2.1 Language-specific Animations .................. 121
    12.2.2 Providing Pedagogical Context by Tailoring .... 121
    12.2.3 Producing and Consuming Online Materials .... 122
  12.3 Learners' Point of View ........................... 122

13. Research Validity ..................................... 125
  13.1 Internal Validity ................................... 125
  13.2 External Validity .................................. 126
  13.3 Technical Considerations ........................... 127

14. Supporting Other Kinds of Processes .................. 129

15. Recommendations .................................. 131

16. Future Work ......................................... 133
  16.1 Technical Development .............................. 133
  16.2 Future Research ................................... 135

A. Built-in Operations in Jsvee .......................... 137

References ............................................. 141

Publications ........................................... 151
List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.


V Peter Brusilovsky, Stephen Edwards, Amruth Kumar, Lauri Malmi, Luciana Benotti, Duane Buck, Petri Ihantola, Rikki Prince, Teemu Sirkiä, Sergey Sosnovsky, Jaime Urquiza, Arto Vihavainen and Michael Wol-


Author’s Contribution

Publication I: “Exploring Expression-level Program Visualization in CS1”

This publication describes the benefits of using expression-level program visualization instead of more traditional line-based visualization. I am the sole author of this publication.

Publication II: “Jsvee & Kelmu: Creating and Tailoring Program Animations for Computing Education”

This manuscript (under revision) describes the technical implementation of the Jsvee library for creating program animations and the Kelmu toolkit for tailoring animations. I am the sole author of the publication and have implemented both tools.

Publication III: “How Do Students Use Program Visualizations within an Interactive Ebook?”

This publication summarizes how students used program animations as an embedded part of CS1 course materials. I implemented all the animations and was responsible for the data collection and analysis. I contributed to reporting on the results.

Publication IV: “Animated Examples as Practice Content in a Java Programming Course”

This publication reports the results of using program animations in a CS1 course using the Java programming language, and compares how the program animations affected the learning results compared with the static code examples which had textual explanation for each line. I implemented
all the animated examples used in the study and was partly responsible for the data collection and production of descriptive statistics about the usage of the animations. The data analysis and reporting are mainly done by the first and third authors. I contributed to reporting on the content I created, statistics I produced, and the future work.

Publication V: “Increasing Adoption of Smart Learning Content for Computer Science Education”

This publication discusses the problems of integrating smart learning content to learning management systems and existing smart learning content in general. I was part of the team that constructed an overall picture of the field before the whole working group convened. Later, I was part of the team which discussed the existing architectures of hosting smart learning content and partly with the team which discussed the problems of integration. I reported on these topics with the other team members.

Publication VI: “Improving Online Learning Activity Interoperability with Acos Server”

This publication discusses how online learning activities can be distributed by hosting them on a content server that separates communication and content. This makes it easier to integrate the same content with multiple learning management systems. Both of the two authors equally designed the architecture, implemented the server, and wrote the publication.

Publication VII: “Tailoring Animations of Example Programs”

This publication conceptualizes the idea of how automatically generated animations could be tailored by instructors to attach a specific pedagogical context to them. We planned the toolkit together with the second author, I designed the implementation, wrote the software, and contributed to reporting on the technical details and figures.
Terms

We use the following technical user roles written with small capitals throughout the thesis:

**Developer** A technical person who implements online learning activities.

**Instructor** A person who decides what kind of online materials should be used in course materials and designs the pedagogical context for them.

An Instructor can be a producer who designs the materials and has a Developer to create them or a consumer who uses materials designed by other Instructors.

**Learner** A person who uses the interactive materials to learn new concepts.

A person can have multiple roles, for example, an Instructor can also be a Developer.

We use the following terms:

**Program Animation** A subcategory of program visualization; a sequence of figures which visualizes how the dynamic state of the program changes when the program is executed.

**Tailoring** A task for Instructors to add a pedagogical context to an automatically generated animation by adding explanatory texts and otherwise customizing the animation for a particular need.

We use the following abbreviations:

**LMS** Learning management system, an online learning platform which hosts course materials and exercises and stores course data such as participants and their grades.

**LTI** Learning Tools Interoperability [46], an interoperability protocol for LMSs to launch external online learning activities and submit grades back to the LMS. LTI is defined by IMS consortium.
1. Introduction

1.1 The Web as a Platform for Learning Resources

The nature of websites has dramatically changed in the recent years. Instead of having only static content, nowadays almost all websites have some interactive elements which take advantage of using modern web browsers and the rich set of features they support. Web browsers provide a flexible platform for developers to create sites that work on many platforms and devices.

The new possibilities of using modern web technologies have also given rise to new ways of creating interactive online learning resources such as interactive animations to explain concepts or quizzes which can give personal feedback based on the answers. Instructors can create online learning environments for their courses which consist of text, interactive materials, and exercises that support learning instead of using traditional course books. However, there are still technical and pedagogical challenges to overcome in order to produce online learning resources.

In this thesis, we discuss the overall process of creating and distributing online learning content and how we can support the process with software. We look at creating new content but also making the content reusable for multiple courses which may use different learning management systems and have different pedagogical approaches. Although we use computer science education and introductory programming as an example throughout, the problems and solutions that we propose are more generic and can be applied to other disciplines, as well.

This thesis addresses three main concerns: namely, the creation, sharing, and customization of learning materials. First, we discuss the problem of creating animations that represent the execution of a computer
program at a suitable level of abstraction so that a novice programmer can understand how the program works and the concepts used in the program. Second, we discuss how the interactive content can be published online to be used in learning management systems and also shared with other instructors. Third, we highlight the need for use of the same content in different pedagogical contexts, and therefore there should be a way to customize the content for a particular use case.

In this thesis, we present three software artifacts to support creating, distributing, and tailoring online learning materials, respectively. Although our viewpoint is primarily technical, we explain how the technical decisions are based on literature, report how these systems have been part of real programming courses to allow for research, and what kind of results we have obtained.

Our motivation is to support the process so that it benefits instructors, developers, and learners. We use these three viewpoints to evaluate the outcomes. We call these viewpoints roles, and in technical contexts, we indicate them in small capitals. The roles are defined as follows:

**DEVELOPER** A technical person who implements online learning activities.

**INSTRUCTOR** A person who decides what kind of online materials should be used in course materials and designs the pedagogical context for them. An INSTRUCTOR can be a producer who designs the materials and has a DEVELOPER to create them or a consumer who uses materials designed by other INSTRUCTORS.

**LEARNER** A person who uses the interactive materials to learn new concepts.

A person can have multiple roles. For example, in some cases the same person can be an INSTRUCTOR and a DEVELOPER.

In the next sections, we give more background on the three problem areas of creating, distributing, and tailoring online content.

### 1.2 Program Animation as an Aid to Learn Programming

Learning new concepts requires work, and learning to program is not an exception. Problems that novice programmers encounter are regularly reported in literature. Reports of common programming-related misconceptions have been reported since in the 1980s (e.g., du Boulay [29], Pea [87])
until the present day (e.g., Shah et al. [101], Grover and Basu [37]). There are multi-national studies which show that novices have problems in writing as well as reading program code (e.g., McCracken et al. [76], Lister et al. [61]). Both skills are vital. But what makes it so hard to learn to program? Are there ways to lower the barrier? How can teachers help students in this process?

One of the main problem sources is the nature of giving formal instructions to a machine which will execute them one by one. When issuing commands to a person, the recipient can query the reasoning behind the commands if they are unclear or seem to be incorrect, but computers do not validate the feasibility as long as the instructions are syntactically correct and executable. If there are, for example, some logical errors in the program code, it might be tough for novices to figure out the error, and understand why the program is not working as expected since the computer cannot suggest “Did you actually mean, for example, the variable x instead of variable y”.

To learn and understand how the program state changes during execution, and to be able to trace the execution mentally, a novice programmer should be able to construct robust mental models of how program execution proceeds in any situation. Du Boulay [29] introduces the concept of a notional machine and defines it as “the general properties of the machine that one is learning to control”. A notional machine is an abstraction of a real computer and its hardware and software which form the execution environment. It does not typically define how the processor, for example, exactly works but gives a way to think of the execution at an abstract level which is suitable for novices, but still realistic without violating the actual model at lower abstraction levels.

For a novice, the logic of computers may feel strange and random because it is not possible to open the cover and see the notional machine in action. Instead, the novice must learn many abstract concepts that define the behavior. At first, program code might look like arbitrary lines of code, but little by little, the novice starts to understand basic concepts such as variables, conditional statements, loops, and how they define the runtime semantics of a program. It becomes more natural that there are certain rules which control program execution.

Du Boulay [30] suggests that teaching this kind of abstract model can be made explicit. It is possible to create a concrete visualization of a notional machine and use the visualization to show how the written static code
Introduction

and the underlying semantics control the notional machine. Instructors can design code examples which will point out important aspects when the program code is visually executed.

This gives the background for Part I which presents the field of program visualization (PV) in more detail. In educational contexts, PV is often used to visualize notional machines and the dynamic aspect of programs with tools such as Jeliot [13] or Online Python Tutor [38]. These tools generate generic program animations which show in step-by-step fashion how the state of a program changes as the execution proceeds. By using these kinds of visual representations, novice programmers have an opportunity to build valid mental models of the execution process and enhance their understanding.

1.3 Distributing Reusable Online Activities

Traditionally, university level courses have used printed textbooks. One of the first ways of publishing interactive content was to provide a CD-ROM together with books. After web technologies had developed enough, it was possible to publish additional interactive materials on webpages. Instructors could create new materials such as animations or quizzes, put them online, and share the link with the students.

Nowadays, more and more materials are online. Learning management systems (LMS) such as Moodle [2] and Blackboard [1] can host a learning environment with all the required content for a course. Instead of using printed books with only static content, modern web technologies can offer the same content digitally and enrich the materials with dynamic and interactive content. Traditional books may even become obsolete at some point in the future if the potential of digital materials is fully utilized.

We identify that there are two kinds of instructors: producers and consumers. There are instructors who are interested in designing interactive materials and activities for learners. For these instructors there should be an easy way to publish the materials online so that it can be easily integrated in multiple LMSs which may use different technical approaches to communicate with external activities.

For the instructors who consumes interactive materials designed by other instructors but do not actively produce content by themselves, selecting the content should be the main focus instead of technical issues, such as how the online activities can be added to the LMS in use.
This kind of community in which the activities are produced and also used outside of their own university can increase the number of available activities and enhance the quality. In the optimal case, instructors can produce activities together so that there are active communities in which each instructor produces a few activities, and in this way, they have materials for a complete course instead of implementing similar activities multiple times separately.

These possibilities and challenges of distributing and creating reusable online activities are discussed in Part II.

1.4 Tailoring Automatically Generated Animations

Program animations can be automatically or semi-automatically generated, for example by giving the source code to a software which then produces an animation. This automation reduces the manual work, such as designing and implementing the animations in PowerPoint without any automated help in the creation process. The same applies for other animations as well.

However, the automatically produced animations typically use the same pre-defined rules for visualizing things, and therefore, the output is generic and may not always be optimal in the pedagogical context where the animation should be used. For instance, in the case of program animations, there can be a lot of steps which are not relevant in a particular context and are present only to form a running example. There should be a way for instructors to tailor the animation so that it would be pedagogically designed for a particular use case. Tailoring means the work to customize an existing generic animation. Tailoring can be, for example, augmenting the existing animation by adding visual annotations such as textual explanations to emphasize the key steps. Tailoring can also customize the stepping of an animation.

When using interactive materials from online repositories, the need for tailoring becomes even more important. Instructors have different approaches to teach and explain concepts. Therefore, instructors would like to use these external activities so that they are aligned with the other content and have the pedagogical context which is relevant for that particular purpose.

We discuss the benefits of tailoring animations in Part III. We refer to the cognitive load theory (e.g., Sweller [112]) and Mayer’s theory of
multimedia learning [71] to give a rationale to why tailoring may enhance learning.

1.5 Supporting the Production Process

We have now shortly described the three main areas of this thesis. If we want to create a new program animation, publish it online, and give a way to tailor the animation, what kind of tools do we need for this? We present in Figure 1.1 a typical process of creating program animations. The process is independent of the systems presented in this thesis, but we will describe how the systems can support this process. The systems are described in Parts I – III.

![Figure 1.1](image)

**Figure 1.1.** The process of creating, distributing, and tailoring program animations. The colored rectangles represent the names of the technical solutions described in each part of this thesis.
We start with the problem of how to create interactive program animations which can be embedded into online course materials. This forms an integral part of this thesis as program animations are our running example of interactive learning content. In Part I, we present Jsvee, a JavaScript library for creating generic program animations. It is an extensible library to implement language-specific animations for online environments.

The process of creating content in Figure 1.1 begins with a case that an instructor has a code example in mind that would be beneficial in course materials. A developer can implement the animation by using the Jsvee library.

When the animation is ready, the next step is to make it available online. If the content does not need any interaction with the LMS such as submitting a grade or information that the animation is viewed, a simple web page with an iframe element is enough to contain the activity. But if the animation should be an activity in the Moodle learning management system [2], for example, and a learner should see a completion mark in the LMS, more advanced technologies are required. Therefore, in Part II, we present the Acos server which can host interactive learning activities and provides means for making the activities reusable in different LMSs.

The final step in this production process is to tailor the content. In Part III, we highlight the importance of having the pedagogical context attached to the activities instead of having fully generic animations. For this purpose, we present the Kelmu toolkit which instructors can use to tailor existing animations by adding explanatory texts and otherwise customize the animation.

Although creating program animations is the main goal for this process in this thesis, Acos and Kelmu are more generic and can be used with other kinds of content as well. Acos and Kelmu are designed to work well in this domain but still to be usable in a wider range of interactive materials.

1.6 Research Questions

In the preceding sections, we have presented an overview of a production process and what kind of software solutions could support the process. In the following sections, we elucidate why also the research point of creating the software is important. As an introduction for the research topics, we
### Figure 1.2. The three main problem areas of this thesis. The questions represent some examples of the challenges and questions each role has in these three steps of the process. The colorized boxes represent the systems to support each part of the process.

<table>
<thead>
<tr>
<th>Learner:</th>
<th>Instructor:</th>
<th>Developer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>How does this code example work?</td>
<td>What kind of program visualizations would be useful for novices?</td>
<td>What can I implement program animations without too much work?</td>
</tr>
<tr>
<td></td>
<td>Where can I find suitable content for my materials?</td>
<td>How can I easily distribute online activities?</td>
</tr>
<tr>
<td></td>
<td>How can I augment and customize the animations?</td>
<td>How can I add pedagogical context to animations?</td>
</tr>
</tbody>
</table>

**Technical solution to the problem:**
- **JSee:** Produces generic expression-level program animations.
- **Acos:** Hosts activities that can be used with multiple LMSs.
- **Kelmu:** Provides tools to tailor animations.

---

**1.6.1 Problems in Program Animation**

After the 1970s, many program visualization tools aimed at novices to teach program execution and notional machines have been developed (Sorva et al. [108]). However, these tools are mainly desktop applications which require installation and a computer with the appropriate operating system. Only few systems are web-based and work in a browser. As more and more learning materials are on the web, it is a natural decision to develop animations which work in a standard web browser to make them platform-independent. The problem is that there is not a generic framework available which would support implementing web-based program animations that can have language-specific animation features.

Because the program visualization tools are often independent projects, developing new features in one tool does not benefit the others. It is also common that a visualization tool supports only a single programming language. Instead of always creating new tools for producing program animations from scratch, it would be useful to have a framework that provides the essential features that are necessary for creating program animations and then extend the features for a particular need. This kind of a common core that new projects could use as a starting point would be potentially...
beneficial for developers and the community.

These requirements bring with them many technical challenges and define the first research question:

**RQ1** How can we help developers to create interactive program animations for multiple programming languages?

a) What kinds of features are needed?
b) How can we design such a toolkit?

This question is explored in Part I and in Publications I – IV.

### 1.6.2 Problems in Distributing and Creating Reusable Online Activities

Having a web page which contains a quiz or an interactive animation is a rather easy task. However, creating content which could be launched from a LMS, identify the learner, and submit some data, such as the grade, back to the LMS requires much more than a simple HTML page. Modern LMSs support having external learning activities integrated in the LMS in a way that a learner does not have to know that the activity is actually hosted in another server because it appears inside the LMS as any other content.

To achieve this interoperability between LMSs and activities, many different protocols to launch activities and allow bidirectional communication exist because of the diverse field of LMSs in use. Some universities have their proprietary LMSs which use a protocol used only in that system. Only recently, more standardized ways to provide communication between the learning management system and the online activities have been published. Nowadays the IMS LTI protocol [46] is supported in most used commercial or open-source LMSs.

Although interoperability protocols such as LTI exist, adopting the protocol is not self-evident. For a developer, the main task is to implement the content and not to worry about the communication protocols and integration. When developing activities, the details of authenticating the launching request, storing some session data, and other details which are related to interoperability protocols should not be part of the task. These details are different for each interoperability protocol. In the best case, a developer would not have to worry about the actual implementation of an interoperability protocol at all as they are often complex.

These problems in reuse and distribution of the activities formulate the
second research question:

**RQ2** How can we help DEVELOPERS to distribute online learning activities and to reuse content in multiple learning management systems?

This question is addressed in Part II and in Publications V and VI.

### 1.6.3 The Need for Tailoring Animations

The process to produce an animation from the given program code can be manual (e.g., working with PowerPoint), semi-automatic (e.g., using jGrasp [45]) or fully automatic (e.g., using Jeliot [13] or UUhistle [109]) depending on the capabilities of the tools. In the case of automatic or semi-automatic tools, it is quite common that INSTRUCTORS have limited opportunities for customizing the generic animations, if it is possible at all.

Automatically or semi-automatically produced animations may contain a high number of steps because they produce a generic animation purely based on the program code. Some tools can automatically generate explanations for the steps but as the tools cannot understand the context of the program, they can produce only generic explanations of the step without explaining what is the purpose of the step for the program at a higher level of abstraction.

Because of these restrictions, animations can expose important details that are not explicitly explained to LEARNERS. Therefore, understanding a generic animation can be difficult, and it should be possible for INSTRUCTORS to tailor animations to add the pedagogical context and explain the important steps in that particular context. This will likely enhance learning as the key content becomes more explicit and highlighted, and animations can be better aligned with the other materials.

The problem is not specific only for program animations, and therefore we will look at this problem to cover all kinds of interactive animations in other disciplines as well. The possibility to tailor animations defines the third research question:

**RQ3** How can we help INSTRUCTORS tailor generic interactive animations to particular pedagogical contexts?

This question is discussed in Part III and in Publications II and VII.
1.7 Relationship with the Publications

Table 1.1 summarizes the relationship with the research questions and publications.

Table 1.1. The relationship with the publications and research questions

<table>
<thead>
<tr>
<th>Publication</th>
<th>RQ1</th>
<th>RQ2</th>
<th>RQ3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication I</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Publication II</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Publication III</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Publication IV</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Publication V</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Publication VI</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Publication VII</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

1.8 Structure of the Thesis

Parts I – III address the three main topics: creating, distributing, and tailoring online activities, respectively. In Part IV, we discuss the contribution of this thesis and its validity. In Part IV, we also give recommendations how Instructors and Developers can benefit from the results we present. The end of Part IV contain ideas for future work to develop the systems further and how to continue the research of quantifying the effects on learning.
Part I

Creating Program Animations

The topic of this first part of the thesis is the field of program visualization and program animation. First, we give definitions for the terms and an overview of the field. In Section 2.3, we shortly present some existing program visualization tools and describe the problems with the current tools. After this, in Chapter 3 we explain the need for a new library to create program animations. In Chapter 4, we introduce the Jsvee library which provides a way to produce interactive program animations for online materials. We discuss the technical implementation, architectural decisions, how it has been used, and what kind of results we have obtained. Our end goal is producing program animations for novice programmers although the field of program visualization has tools for professional software developers, as well.

Relevant publications for Part I are Publications I – IV. The research question for this part is:

**RQ1** How can we help **Developers** to create interactive program animations for multiple programming languages?

a) What kinds of features are needed?

b) How can we design such a toolkit?

Source code and documentation for Jsvee is available at:
https://github.com/Aalto-LeTech/jsvee
2. Related Work

2.1 The Field of Software Visualization

To position this thesis to the appropriate research area, there are two important terms which help to clarify the context: software visualization (SV) and program visualization (PV). In the following sections, we describe the meaning and relationship with these terms and how they are related to this thesis.

2.1.1 Software Visualization

Software visualization is an umbrella term for many kinds of software-related visualizations. Table 2.1 contains a few definitions for the term. However, the same overall goal is included in all of them; software visualization aims at giving understanding of the structure and functionality of the visualized software.

Mattila et al. [65] report in their systematic literature review on software visualization that the three most common reasons for using SV are understanding the software structure, behavior, and evolution of software. Based on the review, the most typical forms of visualization are hierarchical or graph visualizations, geometric projection techniques and text-based visualizations such as tag clouds.

2.1.2 Program Visualization

To narrow the field of software visualization, we refer to the taxonomy by Price et al. [90]. In this taxonomy, software visualization is divided into two subfields: algorithm visualization (AV) and program visualization (PV).
Table 2.1. Various definitions for software visualization.

<table>
<thead>
<tr>
<th>Author</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price et al. [90]</td>
<td><em>Software Visualization is the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software.</em></td>
</tr>
<tr>
<td>Gračanin et al. [36]</td>
<td><em>The field of software visualization investigates approaches and techniques for static and dynamic graphical representations of algorithms, programs (code), and processed data. SV is concerned primarily with the analysis of programs and their development.</em></td>
</tr>
<tr>
<td>Knight and Munro [50]</td>
<td><em>A discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration.</em></td>
</tr>
<tr>
<td>Ball and Eick [9]</td>
<td><em>Software visualization tools use graphical techniques to make software visible by displaying programs, program artifacts, and program behavior. The essential idea is that visual representations can help make understanding software easier.</em></td>
</tr>
</tbody>
</table>

Price et al. define program visualization as “the visualization of actual program code or data structures in either static or dynamic form” whereas algorithm visualization emphasizes a higher-level abstraction of an algorithm which is often written in pseudo-code. Program visualization can be used to visualize an algorithm inside a program, but typically, algorithm visualizations concentrate only on the algorithms. However, the division is not always that obvious as sometimes understanding an algorithm may require following the program code execution very closely.

The taxonomy divides PV further into two subcategories: static and dynamic program visualization. Figure 2.1 by Sorva et al. [108] illustrates the field of software visualization and is loosely based on Price’s taxonomy.

To differentiate the static and dynamic aspect of PV, Ball and Eick [9], for example, define three properties of software which can be visualized: software structure, runtime behavior, and the code itself.

The static aspect of PV visualizes the software structure and code. These kinds of visualizations can, for example, visualize the class hierarchy and
the connections between the classes, how data is transferred between different parts of the program, and how the program code is structured. Static analysis of the source code without running the software is typically the method of retrieving the information for the visualization. Applications or even larger systems can consist of a very large number of classes, even thousands of them. Therefore, this part of program visualization can help professional developers to understand the structure of complex systems. Some program visualization tools, such as BlueJ [10], also have class diagrams aimed at novices.

The dynamic aspect of PV, which we focus on, visualizes the runtime behavior of programs. These visualizations represent in various ways the program state and how it changes when the program is executed. Visualizations can emphasize, for example, data in memory and program flow. A typical debugger is a traditional example of dynamic program visualization. The level of using graphical visualizations varies. Most of the debuggers represent the program state as tables containing memory addresses and other low-level details. However, there are tools aimed at novices, which especially use the graphical representation to clarify concepts such as the call stack and references. The abstraction level and usability are also designed for novices because debuggers tend to be too complicated for novices. We give an introduction to some of these novice-friendly tools in Section 2.3.
2.2 Program Animation

To describe even more specific area of the dynamic PV, we introduce the term program animation. We use the term similarly to Sorva et al. [108] to describe a sequence of visualizations which graphically present how the program state changes during the execution. This can be done with interactive animations so that the changes between the states can be emphasized. These interactive animations are the exact focus of this part of the thesis.

From now on, we only focus on the educational viewpoint of program visualization and animation, although the field of PV also provides important tools for professional developers who use them in their everyday work.

2.2.1 Why to Use Program Animation?

In this section, we provide a rationale for using program visualization by presenting a few viewpoints in what kind of situations program animations can enhance learning, and how program animations can support novices who may have problems in understanding the basics of programming.

Visualization as Learning Aid

Before we discuss the details of using program animation, let us take a broader view to see how visualization is used in education of science, technology, engineering, and mathematics (STEM).

For example, Linn et al. [60] mention visualizations which make unseen processes visible. The phenomena can be too small, complex, long term or large scale to be observed in a normal classroom setting. They give examples such as atomic-scale phenomena including chemical reactions and electricity. Similarly, Chiu and Linn [24] list how dynamic visualizations can illustrate chemical phenomena such as bond breaking and formation.

However, both Linn et al. and Chiu and Linn et al. raise concerns about the effectiveness of the visualizations. They use the term deceptive clarity to describe a situation in which a visualization seems to be easy to understand but a student can recall only superficial features or has overestimated the understanding of the visualization. This same observation is made by Petre [88]. Petre states that graphics do not guarantee clarity and knowing where to look is not obvious.
McElhaney et al. [77] reviewed 47 independent comparisons between dynamic and static materials to estimate the effectiveness of the visualizations in similar settings as already mentioned above. In their definition, dynamic visualization can range from simple animated GIFs to animated slides and complex modeling environments. They report that dynamic visualization offers no advantage to learners for recalling information, but they found out that dynamic visualizations appear to offer an advantage for conceptual learning.

Results by McElhaney et al. [77] suggest that working collaboratively with dynamic visualizations can enhance the learning compared to using them alone. Also, using dynamic visualizations longer (days or weeks) as a part of classroom instruction instead of a short period, for example, in laboratory settings seems to lead to better results.

A recent study by Velázquez-Iturbide et al. [121] reports positive results from an experiment which measured how program visualization affects student motivation. If the results are generalizable to other contexts as well, it means that visualization can enhance learning but also increase student motivation which affects the attitude towards the discipline.

These studies, among others, show that there are many different opportunities to use visualizations as a learning aid. In well-designed learning contexts, visualizations can significantly help in the learning process. Therefore, it is meaningful to use visualization in the field of programming, too.

Understanding Program Execution Model

The main objective for using program animation in introductory programming is to teach the dynamic execution model of programs. Berry and Kölling [15] have separated two purposes: the comprehension of programming and the comprehension of programs. The comprehension of programs aims at understanding how a specific program works whereas the comprehension of programming aims at teaching how programming concepts such as variables, conditionals, loops, and functions work in general. To understand the program execution model, the code examples are chosen so that the visualization can emphasize particular concepts. When program visualization is used for the comprehension of programs, the user is already familiar with the basic concepts, and the purpose is to understand, for example, why the logic in the program does not work.

The purpose for program animation in this thesis is the comprehension
of programming. We use animations to visualize the execution model and how the program behaves when it is executed. This idea is coupled with Du Boulay's [29] concept of a notional machine. It describes the general properties of the machine and software that the novice is learning to control. A notional machine is an abstraction of the actual hardware and software, but it is still accurate enough at the required level of abstraction to describe the dynamic execution process. Notional machines have language-specific features, and even the same programming language can have different kinds of notional machines depending on which language features are required or what is the proper abstraction level.

Sorva [107] has collected notional machine related issues from the literature. He concludes that understanding notional machine is a major challenge for novices, especially the “hidden” parts of the execution which are not clearly present in the program code. These are, for example, references and automatic updates to loop variables. Sorva highlights the importance of explicitly teaching notional machines, and using program animation is one such method. An animation can provide a graphical representation of a notional machine that is capable of executing a piece of code in a particular programming language.

A recent study by Hidalgo et al. [41] organized colloquies as a part of a CS2 course. In these colloquies, students were asked to trace and explain a program which calculates median value. At the same time they explained the execution, they were also instructed to draw a notional machine and use that together with the explanation. They evaluated students’ knowledge of 22 concepts based on the drawings and explanations. Students had excellent mental models in seven evaluated concepts related to very basic concepts such as conditionals, standard output and for loops. However, for nine concepts students had only deficient mental models. These concepts included, for example, function calls, integer division, and local variables. The results also indicate that in this experiment, students struggle with notional machines and have inaccurate mental models. They suggest that tasks in which students draw notional machines and explain the execution can give a better estimate of the knowledge level than traditional exams.

We believe that taking program animations as a part of every day teaching and course materials can increase the understanding of notional machines. Notional machines cannot be learned if they are not made visible, and without this understanding, a novice programmer cannot fully un-
derstand the execution logic which is the most crucial part in the basics of programming.

**Preventing and Correcting Misconceptions**

Research related to misconceptions of novice programmers started already in the early 1970s. We know that there are certain areas in which it is typical that novices do not have viable execution models. These are, for instance, state of the program (e.g., Kurvinen et al. [55]), variables and assignment statements (e.g., Du Boulay [29], Simon [103], Kohn [51]), conditionals (e.g., Bayman and Mayer [11]), function calls with parameter passing and return values (e.g., Madison and Gifford [63], Fleury [32]), and object-oriented programming as a whole (e.g., Holland et al. [43], Ragonis and Mordechai [91], Sanders and Thomas [96]). Program animations can help in constructing these models and resolving these misconceptions.

Pea [87] defined in 1986 the term *superbug* based on his and others observations. Pea describes the superbug as “there is a hidden mind somewhere in the programming language that has intelligent, interpretive powers.” This means that a novice may even ponder that computers are able to think and can help programmers to achieve their goals.

We might think that as computers are nowadays a crucial part of our everyday life, we are somehow better aware of their capabilities. However, Ragonis and Ben-Ari [91] reported in 2005 that there are students who think that some things happen with no cause. They mention misconceptions such as objects are created by themselves, attribute values are updated automatically according to a logical context, and that the system does not allow unreasonable operations. All of these indicate that students do not have a clear vision of how computers work, or what is the role of program code to control the execution.

Clancy [25] lists various sources that may cause misconceptions. These are, for example, analogies with English, thinking assignments as mathematical notation, previous programming experience, recursion, etc. To get a better overview of all kinds of misconceptions, Sorva [106] has compiled “Misconception Catalogue” which is a comprehensive list of 162 different types of misconceptions based on the existing literature and his experiences as an instructor.

Some researchers claim that syntax is irrelevant for misconceptions. For example, Sheil [102] and Spohrer [110] have reported that typically the syntax does not cause problems. However, there are also differing results.
A recent study by Brown and Altadmri [19] used a massive dataset (more than 900 thousand users and almost 100 million compilation events) to identify typical Java programming errors. They constructed 18 mistake categories, of which 11 related to misunderstandings of syntax. The most common compilation error is caused by incorrect or unbalanced brackets. Using assignment operator (\( = \)) instead of comparison operator (\( == \)) is the fourth most common mistake.

If a novice programmer has a misconception, carefully viewing the visualization may expose that the mental model of the novice conflicts with the visualization. In this case, the novice can notice that the original interpretation was flawed and can learn the correct execution model. In this way, we can use program animations to build robust mental models and resolve misconceptions.

Animations to Help with No-Function-In-Structure Principle

De Kleer and Brown present the no-function-in-structure principle [26] together with mental models and how to simulate electrical circuits. They provide a simple example of a circuit that could be used as a doorbell. It contains a battery, a switch, and a coil. They discuss how to simulate the circuit and how its behavior will change if some of its properties change.

The principle states “the rules for specifying the behavior of any constituent part of the overall system can in no way refer, even implicitly, to how the overall system functions.” This means that one should be able to understand how each component individually affects the circuit, and by using that knowledge, build a valid model for the whole circuit.

In the case that the principle is violated, it is likely that a learner can not explain the behavior of the circuit if it is slightly modified. Making assumptions based on how similar circuits typically work is not enough to predict the changed behavior.

This principle can also be applied to learning concepts of programming. When looking at the source code, a novice should be able to identify the “components” of the “circuit”, i.e. the code structures in the program. Knowing that, for example, an index-based for loop is used to iterate a collection, provides only knowledge about the function and not the vital structure if the novice does not understand how the loop variable is exactly used for indexing the elements.

Instead of general principles or typical use cases, a learner should be able to understand that the for loop iterates the elements by using the
Related Work

indices which the loop will generate. If the condition to continue the loop is incorrect, some values are not iterated or the index will be out of the bounds. If the novice can process the purpose of the loop at this level of abstraction, the student probably understands the structure of loops in addition to the function and typical use.

Vainio and Sajaniemi [120] used the no-function-in-structure principle in their research setting. They report that students had problems, for example, to trace the execution of for loops. In their example, a loop was not used in a typical way. They give a strong assumption that this was caused by the confusion between the typical function of a for loop and its structure.

Program animation aims at creating valid mental models. We can visualize, for instance, loops and emphasize how the loop condition work. The motivation is that a novice understands how the concepts (i.e. the structure) work and is then able to apply that knowledge in reading and writing code. Program animations can drill down to some specific concepts to make sure novices gain knowledge and can understand their purpose as a separate “components” to construct larger programs.

Lowering the Barrier of Threshold Concepts

The term threshold concept and the theory behind it are not very old. The research of threshold concepts and their presence have been mainly researched after the 2000s. Threshold concepts are described to be concepts which are hard to learn and they block novices of learning more concepts. Understanding a threshold concept typically changes the way of thinking about the subject.

Mayer and Land [78, 79] describe five characteristics for threshold concepts:

- **Transformative**: Learning the concept produces a significant shift in the perception of a subject
- Probably irreversible: It is unlikely that the concept is forgotten or unlearned
- **Integrative** It creates previously unknow links between familiar concepts
- Possibly often bounded differently in different disciplines
- Potentially troublesome to learn
There has been research and debate (e.g., Bousted et al. [18], Sorva [105], Sanders et al. [94, 95]) whether threshold concepts exist in computer science, are they important, and should they be somehow taken into account in teaching. There is no definitive answer for these questions, but at least some researchers are convinced that threshold concepts also exist in the field of computer science.

Sanders and McCartney [95] have written a review of the threshold concepts in the domain of CS. They have listed over 30 research papers covering various topics which have been proposed as threshold concepts. These contain very fundamental concepts such as assignment, notional machine, object-oriented programming, parameter passing, and program state.

Sorva [105, 106] suggests using visualizations to teach threshold concepts. Also Sanders and McCartney list in their review interventions in which visualization has been used to teach these concepts.

For example, Simon [103] has reported that there is a large share of students who have problems to understand the assignment statement and the sequential execution of a program. These can be threshold concepts which they have not understood properly, and therefore, their success rate is so low.

By providing carefully designed program animations, one can visualize threshold concepts so that the example and the animation together can provide an easy way of thinking about the concept, and give a possibility to go over the barrier and learn more.

2.3 Program Visualization Tools

In this section, we will give an overview of the existing program visualization tools, what kind of features they typically have, and how they work. We have selected a few examples which represent the field well and have features that are relevant for this work, but also share some common limitations which we discuss in this thesis. A much more detailed description of the existing program visualization tools and their history is covered in a review by Sorva et al. [108]. Hidalgo-Céspedes et al. [42] extended the work by Sorva et al. and present six more systems which were created after the first review.

Before discussing the applications and tools of the modern era, let us have a short look at the history. The ancestors of the first program visualization applications were physical boards to manually visualize and
manipulate the state of the program. Mayer [68] used a large physical board in the very first experiments on how program visualization can help with understanding program execution. In Mayer's board, see Figure 2.2, all variable values, inputs, outputs, and the code to be executed were illustrated. To make the visualizations easier to implement and control, the development of automated program visualization tools began in the 1980s.

Figure 2.2. Mayer's [69] way of visualizing program execution in the very first experiments of researching the effectiveness of program visualization.

To classify the program visualization tools, we must define their parameters. Maletic et al. [64] proposed five dimensions to describe the tools: tasks, audience, target, representation, and medium. A working group led by Naps and Rößling [85] defined six levels of engagement which describe the capabilities of the PV tools: no viewing, viewing, responding, changing, constructing, and presenting. In this taxonomy, no viewing means that no visualization is used at all, and at a highest level, the visualization is presented to audience for feedback and discussion. The levels start with the typical cases of only viewing visualizations and moves towards more motivating tasks such as modifying and even creating custom visualizations.

Myller et al. [84] extended the classification by Naps et al. to consist of ten levels: no viewing, viewing, controlled viewing, entering input, responding, changing, modifying, constructing, presenting, and reviewing. Their hypothesis is that increasing the level of engagement between learners and the visualization tool results in a higher positive impact of visualization on the collaboration process.

Both of these engagement taxonomies draw inspiration from Bloom's taxonomy [16] which consists of six levels of cognitive complexity: remembering, understanding, applying, analyzing, synthesizing, and evaluating.
Remembering is the lowest level which means that a learner can remember some things but may not fully understand them. At the highest level, evaluating, the learner is able to make judgments based on the received knowledge. Due to criticism, the original taxonomy published in 1956 was revised in 2001. The revised version [6] is otherwise similar but the two topmost levels are now *evaluating* and *creating*. Creating as the highest level means that the learner is able to apply the knowledge to create new things based on the learned concepts.

Sorva et al. [42] created their engagement taxonomy for classifying the capabilities of program visualization tools in their review. It consists of two axes: *direct engagement* and *content ownership*, see Figure 2.3. Direct engagement describes the activities the tool supports, and content ownership describes the source of the visualized programs.

![Figure 2.3. The two dimensions of the 2DET engagement taxonomy as presented by Sorva et al. [108]](image)

Next, we present some of the existing program visualization tools and their main features. We have chosen a representative set of mature program visualization tools which have the typical set of features and are designed for imperative programming paradigm. The tools we present are Jeliot, UUhistle, ViLLE, and Online Python Tutor. All these produce *generic animations*. By generic animations we mean that the tool takes the code as input and automatically provides the visualization as output so that the user cannot affect the output. We use the 2DET engagement taxonomy to classify the tools.

### 2.3.1 The Jeliot Family

The Jeliot family [13] has been developed since the 1990s. Initially, Eliot was designed to visualize string algorithms written in C. A second version visualized Java programs instead of C programming language and was
Related Work

called Jeliot I. The further iterations are Jeliot 2000 and Jeliot 3.

Jeliot 3 provides an automated program visualization of the given Java program code in expression-level. Figure 2.4 shows an example of the user interface. The user can type the Java program, compile it inside the program and start the visualization. The right-hand side of the window contains the theater which opens the curtains and shows the play when the execution starts.

![Figure 2.4. Jeliot 3 running a short example of using a class and instances. The right hand side of the application is the theater which animates the execution.](image)

The representation emphasizes the expression-level evaluation and the relationship between the objects and references assigned to variables which are visualized as arrows. Only the topmost stack frame is visible. A learner can step forward manually or press the play button and view the animation without pauses. It is not possible to move backward, and because of this starting over is the only option to revisit earlier steps. In terms of 2DET, Jeliot supports controlled viewing.

Jeliot 3 does not provide any textual explanations about the animation steps but can automatically produce multiple choice questions which ask, for example, what is the value that will be assigned to a specific variable. The learner must predict the answer before the execution is shown.

The implementation of Jeliot 3 is based on an intermediate language called MCode [81]. It is designed for this purpose to describe the actions to be visualized. Jeliot uses an internal Java runtime environment which is modified to produce MCode while the code is executed. Jeliot receives the produced MCode instructions and animates the program execution accordingly.

Currently, Java is still the only supported language. However, the orig-
inal idea of MCode is that the instructions are not language-specific, and therefore, Jeliot could also visualize other programming languages if there was a way to produce MCode instructions for programs written in other programming languages than Java.

The Jeliot family has been a part of many research settings in Finland and other countries. Ben-Ari et al. [13] summarize the results. They have published several publications which show results that using Jeliot enhances learning when novices have used the tool as a part of their learning process.

Jeliot has also been used to create conflicting animations [80]. The idea is to motivate learners to follow the animation and pay attention when the animation does not proceed as it should. They report results that using conflictive animations improves learners’ metacognitive skills and their conceptual knowledge improves at better rate if compared with those learners who used a standard version of Jeliot 3.

2.3.2 UUhistle

UUhistle [106] is a program visualization and simulation tool for Python programs. UUhistle is capable of producing automatically a visualization of the given program code. Figure 2.5 shows an example of the user interface. The language features UUhistle support are limited but cover the typical CS1 topics, such as arithmetics, variables, control statements, functions, recursion, lists, dictionaries, classes, instances, etc. In terms of 2DET, UUhistle supports own content and responding.

The visualization emphasizes the stack frames, local variables and especially the expression evaluation in the yellow evaluation areas in which all the expressions are evaluated. UUhistle does not show the reference arrows as in Jeliot but they become visible when hovering the mouse over a reference. The division between the stack and heap represents the idea of references in object-oriented programs. The local variable contains only a reference to a memory location which holds the actual object and it can also visualize that there can be multiple references to the same memory location, and they all refer to the same object.

In addition to the traditional program visualization, UUhistle provides a novel exercise type of visual program simulation. Visual program simulation as presented by Sorva [106] motivates the learner to constantly think the execution as the learner must drag, drop, and click the graphical elements in the same way as the animation would proceed. The learner sim-
Internally UUhistle uses the Jython framework \(^1\) which is a Python implementation written in the Java programming language. UUhistle uses Jython for both parsing and executing parts of the Python code. Parsing the given program code produces an abstract syntax tree (AST), and the elements in the AST are converted to UUhistle's intermediate language to be executed. UUhistle takes care of the control flow but uses the Jython framework to evaluate expressions. Therefore, the code examples can quite freely use the basic API of Python and those data structures which UUhistle supports.

To deepen the understanding of the animations, UUhistle automatically provides explanations for each animation step. In addition to a short description in the status area, a novice can also open a longer manually written description for that type of steps which explains the purpose of the step in more detail. These explanations can also contain dynamic sections that are modified based on the actual step. Explanations can have predefined variations, and UUhistle automatically selects which version of the text it will show.

UUhistle also features an interactive mode in which UUhistle behaves as a REPL (read-evaluate-print loop) by visualizing the code as it is typed.

\(^1\)http://www.jython.org/
This feature can be used, for example, in lectures where an instructor can run and demonstrate code examples on the fly instead of having all the examples written down beforehand. Also, learners can just type some code and interactively try various operations.

The author’s Master’s thesis [104] contains results of possible misconceptions learners have had while solving the simulation exercises. The recognized potential misconceptions related to, for instance, assignment, function calls, and object-oriented programming covers the topics in the existing literature of misconceptions well.

2.3.3 ViLLE

ViLLE [92] is an application for visualizing program snippets written in many programming languages. ViLLE has a collection of sample programs which learners can explore, and an instructor can also create more examples.

The main user interface of ViLLE can be seen in Figure 2.6. It consists of the control buttons and the view which shows the code, stack frames, and local variables inside them. ViLLE can also show textual descriptions for each step.

![Figure 2.6. The user interface of ViLLE program visualization tool.](image)

One of the main features in ViLLE is that it can show the same code in multiple programming languages. This is based on conversion rules which define direct mappings between Java and the other language. Therefore ViLLE can automatically convert the given code to other supported programming languages, and the learner can select which programming language to use. ViLLE supports an intersection of the language features,
mainly arithmetics, variables and function calls. However, this conversion from language to another and working in that intersection means that it cannot support language-specific features which are not part of the intersection. This limits the usage of the tool.

ViLLE uses the line-based visualization. It can also show similar multiple choice questions to Jeliot 3 in order to challenge students to think about the execution beforehand. In terms of 2DET, ViLLE supports given content and responding to the questions.

Nowadays ViLLE is more than a program visualization tool. It is a learning management system for programming, mathematics, physics, languages, etc. Program visualization is not the purview of ViLLE anymore but the web-based version has some new exercise types such as program simulation tasks which increases the direct engagement to the level of applying.

2.3.4 Online Python Tutor

Online Python Tutor [38] is a web-based program visualization tool originally developed for Python, but nowadays supports other programming languages, as well. A learner can type code and then visualize the execution. An example is shown in Figure 2.7.

The implementation of Online Python Tutor is based on the server-side
service which executes the given code. During the execution process, the execution is traced, and the trace will contain all the details the animation will visualize. The real execution environment allows using practically all the language features, and the visualization is only a playback of the execution. It is possible to implement support for multiple languages by generating similar traces that the original Python backend produces. In 2DET, Online Python Tutor supports own content with controlled viewing.

2.3.5 Summary

Table 2.2 summarizes the presented program visualization tools and their features. As we can see, there is not a web-based tool which supports multiple programming languages and expression-level evaluation. This leads us to the next chapter which describes the need for this combination.

Table 2.2. Summary of the presented tools and their features.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Platform</th>
<th>Granularity</th>
<th>Multiple Languages</th>
<th>Direct engagement</th>
<th>Content ownership</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jeliot 3</td>
<td>desktop</td>
<td>expr.</td>
<td>no</td>
<td>ctrl’d viewing</td>
<td>own</td>
</tr>
<tr>
<td>Uuhistle</td>
<td>desktop</td>
<td>expr.</td>
<td>no</td>
<td>applying</td>
<td>own</td>
</tr>
<tr>
<td>ViLLE</td>
<td>desktop / web</td>
<td>line</td>
<td>yes</td>
<td>responding (applying)$^a$</td>
<td>given</td>
</tr>
<tr>
<td>Online Python Tutor</td>
<td>web</td>
<td>line</td>
<td>yes</td>
<td>ctrl’d viewing</td>
<td>own</td>
</tr>
</tbody>
</table>

$^a$Web version only
3. The Need and Aims for the Library

This section describes the need for implementing the Jsvee library presented in Chapter 4 by raising some issues that the current tools have. There is no library or a visualization tool available which would currently fulfill all of the requirements of supporting multiple programming languages, having expression-level visualization and be embeddable in web pages.

3.1 Why a Library Instead of an Application?

There are certain differences between complete applications and libraries. Typically, an application is a working solution which does not need any additional work to achieve the goals that it is designed for. For example, in program animation, INSTRUCTOR can input any code and the application automatically produces the visualization.

Instead of a complete application, a library gives ready-made pieces which can be used to construct different kinds of applications. It depends on the library how much work it requires to construct something meaningful. Although it always requires some developing to produce something with the library, it provides much more freedom of use and features that can be attached to the end product.

The main reason for designing a library is that we want to implement a toolkit which provides the common parts that program animations need. A library is a natural choice to pack these features and give it to DEVELOPERS. If we can create a working library, new program animations can be based on the library instead of implementing all the functionality separately to each program visualization tool.

This kind of approach of providing a library for producing visualizations is not unique. For example, the JSAV [48] library has a similar approach.
The library contains mechanisms that can be used to generate algorithm visualizations and algorithm simulation exercises. It can create all the elements which visualize the data structures, but a developer must define what elements are needed and how the visualization proceeds, or how the exercise should be solved and graded.

3.2 Multiple Programming Languages

Many of the existing program visualization tools support only single programming language. This is often a technical decision based on the implementation details.

If the visualization tool is implemented with the same language as the visualized program code, the visualization tool can efficiently use, for example, parsing tools and debugging interfaces offered by the programming language core API. Also using the same programming language for visualization and implementation provides the major benefit that the visualized program can use the built-in functions the programming language offers. Typically, code examples contain some library functions, and in this case, a real runtime environment is available to execute them.

If the visualization tool is implemented in another programming language than the tool visualizes, there must be ways to provide enough information for the visualization about the program behavior. For example, in Online Python Tutor [38], there is a server-side process which uses the real execution environment and then provides a language-independent trace of the execution which the animation engine can visualize. With this approach, the server-side implementation can trace the execution and collect data when it executes the program, and there is a separate visualization interface which can interpret the trace and visualize it afterwards. However, this prevents visualizing programs which interact with the user as the execution path is often based on the input.

The multi-language support in Online Python Tutor is based on the trace file. If it looks the same regardless of the programming language, the same visualization engine can visualize it. However, as a drawback, this mechanism easily limits the features to be visualized to the most common features, because one visualization must suffice for all the supported languages. There should be a mechanism to provide language-specific features although a common core is in use.

By using a visualization library which supports multiple programming
languages and language-specific features, a Developer can with very little effort use the same library to produce visualizations in several programming languages. The Developer does not have to learn to use different tools or libraries with different programming languages. Still, the most important benefit is that new visualization features can be added to the library to support concepts which exist only in some languages.

3.3 Expression-Level Visualization

The stepping granularity in program visualization tools is typically the same as in traditional debuggers. The program execution proceeds in line-by-line fashion, and the internal structure of the code line is not visualized. Furthermore, an abstraction level which visualized Assembler instructions or register values inside the processor would be too detailed and not relevant in this educational context for novice programmers.

3.3.1 Granularity of Steps in Animation

Program animations typically use two different visualization granularities. These are line-based visualization and expression-level visualization. Expressions are the very basic construction blocks in the source code. For example, assignment statements, function parameters, if statements etc. all contain expressions. Therefore, it is an integral concept to understand that evaluation is constantly required during the execution, and complex expressions can be present in any part of the code, not only in assignment statements, for instance.

In line-based visualization, executing one line is the smallest step the visualization provides. For example, if the line contains an assignment statement \( b = 2 + 3 \times \sqrt{a \times 2} \), we only see the result in the variable \( b \), and it is not possible to see the order in which the expression is evaluated and how the right-hand side of the assignment statement got its value.

The idea and reasons to use expression-level visualization instead of line-based visualization are discussed in Publication I. The main argument for using expression-level visualization is the concept of expression evaluation and understanding that a code line has an internal structure. This is also related to notional machines discussed earlier in Section 2.2.1.

In the previous example of an assignment statement, the right-hand
side is an expression which consists of expressions. For example, a \( a^2 \) inside the \( \sqrt{\text{}} \) function is also an expression which must be evaluated in order to define a value for the square root.

In line-based visualization, it is possible that a novice student thinks that running a line of code is one atomic action and it does not have any relevant internal structure. However, the concept of evaluating expressions is a crucial part of understanding the correct execution model. Learning the expression evaluation and the correct order helps to understand many details that we present in the following section.

### 3.3.2 Examples of Expression-Level Visualization

The expression \( \text{max}(5, \text{abs}(2 \times -3)) \) contains two function calls that are nested. This means that the result of the \( \text{abs} \) function call will be the second parameter for the \( \text{max} \) function call. Figure 3.1 shows how the expression is first constructed and then evaluated step-by-step following the correct order: from left to right, from inner parts to outer parts.

![Figure 3.1. A diagram of evaluating expression \( \text{max}(5, \text{abs}(2 \times -3)) \) in expression-level as shown in UUhistle and Jsvee animations.](image)

Expression-level visualization highlights the evaluation order and can visualize important concepts related to functions such as parameter passing and return values. In the example, we can see that a function call cannot be started before all the parameters are evaluated. We also see that the parameter values will be simple values after the expression is evaluated. There are many misconceptions about parameter values but this kind of visualization can show that they are simple literal values after the evaluation. The idea of how the calls in that example are sequentially executed can also be a threshold concept which prevents understanding this kind of complex expressions but can be resolved with an animation.

In object-oriented programming, expression-level visualization can show step-by-step how memory is allocated for a new object, how instance vari-
ables are created, and how constructor evaluates the initial values for the instance variables and assigns them to the instance variables. It is possible to show much more detailed steps as in line-based visualization.

Expression-level visualization also highlights the internal structure of a line of code. A line consists of smaller building blocks which can be used to construct the desired action for that line. It is important to understand the purpose of each part in the expression to apply similar language constructs in other contexts as well. If a learner only understands the result of the line but not the intermediate steps to achieve the result, applying the same language constructs in other contexts becomes difficult. The previously discussed no-function-in-structure principle in Section 2.2.1 emphasizes this requirement to understand the purpose of each component individually, the parts of the expression in this case.

3.3.3 Teaching Expression Evaluation in Textbooks

Although expressions and the previously mentioned concepts to evaluate them are important, in many CS1 courses they get only a little attention. To see how well these concepts are taught in textbooks, we reviewed some typical CS1 books aimed at novice programmers.

We looked at how the concept of expressions are explained, how evaluating of expressions is described, are there examples of function calls with complex expressions as parameters, and do they explain parameter passing. By complex parameters we mean that there is a call in which the parameter is not a single literal value or variable name but contains a more complex expression.

The review is done so that the explanations are identified at the same point in the book in which the topic was introduced. If there are examples later in the book which use the feature or describe it better, that is not counted. The reviewed books are listed in Table 3.1.

Practically, all the books follow the same order of presenting the basic concepts. The term expression is typically introduced when the assignment statement and arithmetic operators are explained. Book D is the only book which does not clearly define the meaning of an expression.

Evaluating an expression is explained in these books when discussing arithmetic and operator precedence. They give examples of expressions with multiple operators and show in which order the expression is evaluated. However, many books fail to explain that similar expressions can be used anywhere in the code in addition to assignment statements. For
Table 3.1. The books we analyzed how well they explained concepts such as expression, evaluation, parameter passing, and using complex statements in function calls.

<table>
<thead>
<tr>
<th>Abbrv.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book A</td>
<td>Charatan &amp; Kans</td>
<td>Java in Two Semesters [23]</td>
</tr>
<tr>
<td>Book B</td>
<td>Deitel &amp; Deitel</td>
<td>C++ How to Program [27]</td>
</tr>
<tr>
<td>Book C</td>
<td>Deitel &amp; Deitel</td>
<td>Java How to Program [28]</td>
</tr>
<tr>
<td>Book D</td>
<td>Gaddis</td>
<td>Starting Out with Java [34]</td>
</tr>
<tr>
<td>Book E</td>
<td>Gaddis</td>
<td>Starting Out with Python [33]</td>
</tr>
<tr>
<td>Book F</td>
<td>Hanly &amp; Koffman</td>
<td>Problem Solving and Program Design in C [39]</td>
</tr>
<tr>
<td>Book G</td>
<td>Horstmann</td>
<td>Java Concepts [44]</td>
</tr>
<tr>
<td>Book H</td>
<td>Lewis</td>
<td>Introduction to the Art of Programming Using Scala [58]</td>
</tr>
<tr>
<td>Book I</td>
<td>Lewis &amp; Loftus</td>
<td>Java Software Solutions [57]</td>
</tr>
<tr>
<td>Book J</td>
<td>Liang</td>
<td>Introduction to Java Programming [59]</td>
</tr>
<tr>
<td>Book K</td>
<td>Roberts</td>
<td>The Art &amp; Science of Java [93]</td>
</tr>
<tr>
<td>Book L</td>
<td>Savitch</td>
<td>Absolute C++ [97]</td>
</tr>
<tr>
<td>Book M</td>
<td>Savitch</td>
<td>Absolute Java [98]</td>
</tr>
<tr>
<td>Book N</td>
<td>Schneider</td>
<td>An Introduction to Programming Using Python [99]</td>
</tr>
<tr>
<td>Book O</td>
<td>Wu</td>
<td>An Introduction to Object-Oriented Programming with Java [123]</td>
</tr>
</tbody>
</table>

example, books A, E, G, J, L, M, and O do not give examples of function calls which have more than a single literal or variable as a parameter. Because of the simple examples, it may lead to a wrong assumption that a parameter must be a single value and it cannot be an expression that is evaluated.

All books except G and H describe parameter passing, and often there is a section (depending on the language) describing the call-by-value behavior which explains that the passed parameter must be a single value.

Only books D and N give an example of using nested functions and how the concept works. Book N uses nested calls in examples to directly type convert some user input, but fails to explain what that means or how it works.

The review shows that by reading textbooks or using line-based visualization, the understanding of evaluating expressions may be limited. It is
possible to think that, for example, only assignment statements can contain complex expressions and the expression evaluation is related only to arithmetic operations. The expression evaluation is not emphasized in many cases enough so that students would understand that every expression can be as complex as needed. The books do not highlight the possibility to combine and create more complex expressions when needed. If books always use very simple examples to demonstrate new concepts, it does not develop a mental model for a large number of language constructs that can be combined in unlimited ways together to produce the required expression.

Expression-level visualization explicitly shows that the evaluation process is always present, requiring of course that the examples are carefully designed to contain complex expressions.

3.3.4 Teaching Expression Evaluation with Online Tutors

Problets\(^1\) are small tracing tasks for introductory programming to teach the basic concepts. These tasks are solved with an online tutor which gives feedback.

There are Problets for training expression evaluation with arithmetic, relational, logical, assignment, or bitwise expressions. These are available in C++, Java, C# and partly in Visual Basic. In the assignments, the student must select in order which part of the expression is evaluated and what is the result of the evaluation. The tutor gives constant feedback. After the task is solved, an explanation is shown why the execution occurs in that order. An example of the feedback and interface is shown in Figure 3.2 below.

The green curly brackets visualize the sub-expressions and are evaluated in the drawn order from top to bottom. If the student chooses a wrong expression or evaluates the value incorrectly, a red symbol would indicate that instead of the green ticks which are shown in the figure.

By training the expressions, the operator precedence should become explicit, and students should understand how expressions are evaluated. Kumar [52, 53, 54] reported positive results of using Problets when teaching expression evaluation. In the experiments, students practiced the evaluation of arithmetic expressions. Visualization helped students learn significantly more concepts. However, visualizations may be more ben-\(^{\text{http://problets.org}}\)
The Need and Aims for the Library

Figure 3.2. An example of Problets. After the student has correctly indicated the correct order of evaluating the sub-expressions and entered their result, the online tutor will explain the example.

efficient for some student groups than others. For example, less-prepared students seemed to benefit from the visualizations more than others.

3.3.5 Technical Challenges in Visualizing Expression-Level Evaluation

One of the major technical problems in the expression-level visualization is getting enough detailed information about the execution required for the expression-level visualization. Typically, debugging interfaces expose details only in line-based level because most of the debuggers work at this level. Therefore, there is no need to trace the execution with a higher precision and provide information that debuggers do not use.

To overcome the restrictions of debugging interfaces, Jeliot 3 uses an internal Java execution environment that is able to produce more detailed execution traces which contain expression-level details. The Labster tool presented by Juett [47] uses a similar approach of having a tightly integrated custom environment to execute C++ programs to produce enough details for the expression-level visualization that the tool provides.

The third tool which supports expression-level visualization for Python programs is Thonny [7]. It uses the standard Python debugging interface for tracing but adds certain markers to the code before it gets executed. The code to be visualized is analyzed before the execution and automatically modified to contain markers. When the execution environment executes the original code with markers, which can be, for example, function calls at a certain point, Thonny registers the call and can trace the execution order at the required level. After the program is executed with the markers, the execution has produced a detailed trace which can be visualized.

5 / 7 returns 0
Since both the operands are integers, integer division is performed. Any fraction in the result is discarded.
5 % 7 returns 5, the remainder when 5 is divided by 7 using ordinary long division
0 + 5 returns 5
The Jeliot approach of using a custom execution environment is tightly coupled with the Java programming language as the environment itself is modified. The marker approach is more language-independent but modifying the code automatically requires programming language specific parsers and the position of markers must be based on the syntax of the language. Therefore, whatever approach is used to generate enough information for expression-level visualization, the implementation is always somehow programming language specific, and it is hard to develop more generic solutions for the problem.

In addition to the technical challenges, it is also a visual challenge to design how all the different actions in expression-level should be visualized to show the intermediate steps. Therefore, the line-based visualization is easier because it just refreshes the variable values and other elements after each code line.

### 3.4 Embeddable Content for Web Pages

Many of the existing program visualization tools are desktop applications which must be installed. These applications may only run in Windows, for example. Installing new software can be restricted by administrators, and depending on the local policies, this can prevent using the tools, or at least cause additional work for the IT staff members.

Nowadays more and more online materials exist as they provide a completely different level of interaction, and other features which traditional paper books cannot offer. Therefore, it is a natural choice that also the program animations should be part of the online materials. Modern web browsers also provide an efficient platform to implement interactive materials that can be used with a wide range of devices.

Online Python Tutor is one of the few existing program visualization systems which have been developed during the era of modern web applications. The visualizations made with Online Python Tutor can be embedded in any webpage. In this way, a learner can browse the materials in a browser and use the attached visualizations without any additional software or plug-ins.

One of the challenges is the browser environment which can natively execute only JavaScript code and has very strict security policies. Therefore, it is not possible to run, for example, a C++ program in a real runtime environment. To overcome these issues, the user interface can be a web...
application, but it requires a server-side environment which can execute the code in a real environment and then send a trace back to the browser. Online Python Tutor uses this kind of approach. Another option is to use JavaScript to emulate the target language without using a real execution environment at all.

3.5 Features the Library Should Contain

If we consider the previously presented program visualization tools in Section 2.3 and what kind of typical features they share, we can define the minimum set of features this kind of library should provide. If the library should be a building block for easily creating program animations and even more complex applications, it should contain all the basic features which are required to show program animations.

In program animations, various graphical elements such as the stack, variables, heap and so on are present in most of the applications. The library should be able to create these elements and provide a graphical interface for the animation. It saves the DEVELOPERS a great deal of work if they do not have to design the appearance and define how each element moves when the animation proceeds. Still, there should be a mechanism to override the visual appearance if a DEVELOPER wants to change it. All in all, the ability to provide means for the graphical user interface is a clear requirement for the library.

All program visualization applications provide some means to control the visualization. The library should contain an engine which controls the visualization and is capable of moving forward and backward. In the optimal case, the DEVELOPER does not have to worry about the execution if it is specified how the animation should proceed. Not all the tools support moving backward but as it is possible that LEARNERS would like to see some steps again, the engine should support it. Again, these are features which are not relevant for each DEVELOPER to develop separately, and therefore, it is the job of the library to provide means for controlling the animation.

Visualization tools manipulate the program state when the execution proceeds. All programming languages share some general characteristics such as the ability to evaluate expressions, assign values to variables, etc. Therefore, the library should support visualizing those language features which are covered in typical CS1 courses. Tew and Guzdial [116] gathered
a collection of topics which are common in the CS1 textbooks regardless of
the programming language. The list they collected became the so-called
FCS1 concept inventory [117] which contain language-independent con-
cepts that CS1 courses typically cover, and an instrument to measure the
knowledge level of students. That is not the only initiative to create such
concept inventories or a way to find out a the current level of student’s
knowledge. For example, Caceffo et al. [20] as well as Parker et al. [86]
have developed similar concept inventories.

These inventories can be used to check what kind of concepts CS1 courses
teach and make sure that the library is capable of viewing such program
animations which teach these concepts.

Combining these requirements with the previously introduced needs to
present program animations in multiple programming languages, have
support for expression-level visualization and produce embeddable con-
tent for web pages, we have quite a broad set of features the library should
contain.
4. Jsvee Library

This section presents the Jsvee library, its design goals and implementation. These arguments are discussed in Publication II, as well. We also report how the library has been used in CS1 courses, what kind of research we have done with Jsvee, and what kind of results we have obtained.

4.1 Design Principles

Jsvee is a JavaScript library which is able to produce expression-level program visualizations. The library exists because we identified three issues with the current visualization systems. These issues are described in Chapter 3 but for a summary, currently there is not, to our best knowledge, a library for creating program animations which supports language-specific visualizations for multiple programming languages, expression-level visualization, and embeddable animations for websites. This thesis finds ways to create a library which can overcome these issues.

As already discussed in Section 2.2.1, Berry and Kölling [15] use the terms the comprehension of programming and the comprehension of programs. Jsvee especially aims at improving the comprehension of programming.

To get an overview of how the animations made with the Jsvee library look like, an example of an animation created with Jsvee is shown in Figure 4.1.

The visual appearance of the graphics is derived from UUhistle [109]. Although the appearance looks the same, Jsvee does not share any code with UUhistle. Only the ideas and design principles in UUhistle are developed further with a completely new technical implementation.

Whereas UUhistle is a desktop application¹, Jsvee is a library for language-

¹Technically UUhistle supports Java Web Start but is still a desktop application
**Jsvee Library**

**Figure 4.1.** A program animation made with the Jsvee library. The red numbers are added to help with the explanation: ① shows the code and the current line. ② is the topmost stack frame with two local variables and an expression to be evaluated. ③ is the stack frame that initiated the function call running in ②. Area ④ contains functions and operators which are copied to the evaluation area when the expressions are constructed. ⑤ is an automatically generated explanation for the previous step, and ⑥ contains the buttons to control the animation.

independent visualizations and cannot therefore use the Python-specific code UUhistle uses. Jsvee can be seen as a modern way of creating similar visualizations that UUhistle is capable of creating, but Jsvee has a broader use context. Jsvee does not support visual program simulation (VPS) as UUhistle does. But because Jsvee is a library, it has been designed so that it is possible to implement that feature later if needed. VPS has not been in the scope of this thesis.

The visual appearance of the animations made with Jsvee emphasize the stack, stack frames, local variables and especially the expressions which are constructed and evaluated in the yellow areas inside each stack frame. Figures 3.1 and 4.4 show examples of expression-level evaluation in practice. Jsvee is meant for visualizing programs written in imperative programming paradigm. Other paradigms such as logic programming would require totally different way for visualizing the execution. However, functional programs as a part of declarative programming paradigm can be visualized to some extent with the same representation as imperative programs if they are not highly recursive and a multi-paradigm programming language such as Python or Scala is used.

although it can be easily started in the browser
4.2 Supported Programming Concepts

Jsvee contains over 50 ready-made operations which form the different animation steps it can visualize and execute. These operations can be seen as the instructions for the notional machine Jsvee provides. Operations cover the typical concepts which are required in CS1 courses. These operations cover well the typical concepts that Tew and Guzdial [116] list for CS1 courses. We use the same categorization to describe below what kind of typical CS1 concepts can be visualized with Jsvee.

**Fundamentals**  Jsvee can visualize basic concepts such as arithmetic expressions, mathematical operators, relational operators, logical operators, evaluating expressions in the correct order (operator precedence), variables, and assignment statements. These are the first topics in every CS1 course.

**Selection statement**  All the typical if statements with or without else statements can be part of the visualized programs.

**Definite loops**  Jsvee provides a mechanism to iterate integers in a specific range or iterate elements in a collection. These cover the typical for and for each loops. Iterators are not visualized.

**Indefinite loops**  Based on the original categorization, this concept contains while loops although they are not always indefinite. However, Jsvee supports any kind of while loops. It is also possible to visualize break and continue statements.

**Arrays**  Jsvee supports initializing arrays by defining the size of the array or defining and evaluating the initial values. It is possible to fetch as well as assign values by using the index of the element. Arrays are shown as objects with references, and therefore, this topic suites well for introducing the concept references and mutating an array by using a reference.

**Function/method parameters**  Jsvee supports function and method calls that can have parameters. In addition to visualizing the use of calls, visualizing parameter evaluation and parameter passing are one of the concepts which are traditionally hard to understand. Function calls also visualize the stack and stack frames with local variables.

**Function/method return values**  Calls can return a return value to the stack frame in which the call started. Jsvee can visualize how the return value is evaluated, how the value is returned to be a part of the expression in
the lower frame and the topmost stack frame can be disposed at this moment.

**Recursion** Jsvee does not have any restrictions what kind of function calls can be made and how many active calls there can be. Therefore, visualizing a recursive program can be visualized as any program that have function calls. Of course, the level of recursion must be suitable to avoid a very high number of visualization steps.

**Object-oriented basics** Jsvee supports all the basic object-oriented concepts such as having classes and methods, instantiating instances, having multiple instances from the same class, instance variables, and class variables. Also, the this or self variable can be made explicit to have a reference to the correct instance. Jsvee shows references as memory-addresses (simple integers), and pointing a mouse over a reference draws an arrow pointing to the correct instance.

In Appendix A, there is a complete list of the ready-made operations. A smart content server hosted by Aalto University at http://acos.cs.hut.fi contain a number of Jsvee examples written in Python, Java, and Scala to be used in CS1 course materials but as well to demonstrate the visualization features of Jsvee.

### 4.3 Technical Implementation

The technical implementation of Jsvee is detailed in Publication II. Jsvee is implemented in JavaScript, and it uses the jQuery library but no other JavaScript libraries. The visual appearance is defined by a CSS stylesheet as all the elements are standard HTML elements such as divs. Because of this, the browser will place all the elements based on the CSS rules, and the library does not have to draw or position the elements. Jsvee takes care of all the graphical elements, and a DEVELOPER only defines which layout is used to create the user interface. Jsvee provides a few options how the memory areas and other elements are positioned.

The library consists of various internal parts which are illustrated in Figure 4.2. In addition to the internal parts, two additional external parts, a language pack and animation definitions are required as well. To control and use the library, the DEVELOPER must inject the animation

---

2https://jquery.com/
sequence to the library and specify the language pack which can execute (or more likely emulate) the required language-specific features. With external plugins it is possible to extend the library.

![Diagram of Jsvee Library modules](image)

**Figure 4.2.** The internal structure of the Jsvee library and the external parts. External animation definitions and a language pack are required in order to use the library and create visualizations. Plugins can extend or customize the functionality.

The purpose of each module is described in below:

- **Core** Orchestrates all the modules and contains the engine to select the next animation operation and execute it. The core module implements the undo feature and offers hooks to other modules which can modify the state before or after each step.

- **Operations** Provides all the ready-made operations to manipulate the notional machine. Each time when the animation steps forward, the core module reads the next operation in the operation sequence and calls the corresponding function in this module to manipulate the state.

- **Transitions** Defines how to animate a state transition. For example, when an assignment operation is executed, this module defines how the value to be assigned moves from the evaluation area to the target variable.

- **Explanations** Contains a textual explanation for each operation. Explanations are mainly descriptive names for each operation but can have dynamically generated parts such as the target variable.

- **User Interface** Generates the user interface for the animation and provides utility functions for operations to manipulate the UI.

- **Animation Definitions** Defines the operation sequences for the animations which are available. The core module reads the definitions for the re-
quested animation and starts executing it.

**Language Pack** Provides all the language-specific features. Whenever an operator or a built-in function is evaluated, the language pack must produce the result. It can also extend the existing set of operations by defining new language-specific operations.

**Plugins** Extends the basic functionality by providing custom operations which can, for example, implement multiple choice questions or customize the UI. Typically uses the hooks the core module provides.

Although Jsvee itself is language-independent, language packs make it possible to create language-specific visualizations. For example, in Scala programming language there is a concept of companion objects. The Scala language pack can in this case provide the required functionality and new operations which are required to visualize the concept. In this way, the ready-made operations Jsvee provide can be used to visualize the common parts of programming languages in a same way as in ViLLE or Online Python Tutor but the language packs can extend the feature set and add the missing language-specific operations and visualizations which is not possible in those two tools.

The main tasks for language packs is to evaluate operators and built-in functions. A language pack is a JavaScript library that registers hooks to the Jsvee’s core module. When there are operations which require the language pack, the core module calls the language pack and requests the result for the current operation, for example, what is the result for operation $2 + 2$ and what is the data type of the result. In many cases, it is possible to emulate the library functions in JavaScript and use the eval function of JavaScript to evaluate operators. Therefore, emulating the whole language is not necessary to animate typical examples.

### 4.4 Producing Visualizations with Jsvee

The procedure to create a new program visualization is shortly described in Publication II. This section gives an overview of the process. The process is outlined in Figure 4.3. The process starts when an Instructor has a need for a new program animation. The Developer will go through the process and creates the animation.
Step 1: Creating the Operation Sequence

The main challenge with Jsvee is that it does not contain a real execution environment for any programming language. Jsvee does not actually run the code which is shown for the user. Instead, it can execute a sequence of animated operations which change the program state in the same manner as the original code does.

To create a new visualization, the program code must be transformed into a sequence of operations which describes the actual animation steps. This process maps the semantics of the program code with the operations that Jsvee provides. An example of a sequence to animate evaluation of an expression is shown in Figure 4.4.

There are mainly two options how to produce the operation sequence: It can be made manually or using a transpiler which recognizes the program structure and creates the corresponding operation sequence based purely on static analysis of the code. However, as even simple expressions contain many steps, producing these operation sequences for a small program require a lot of careful work. A complete list of the available operations are in Appendix A.

We have implemented a transpiler for Python which automatically parses the code and generates the corresponding animation sequence. Figure 4.4 also shows the abstract syntax tree for the expression and how the operation sequence is produced by visiting the nodes in the tree.

By combining Jsvee library, the Python language pack and the transpiler, it is possible to create an application which is able to show a limited set of Python visualizations depending on the features the transpiler and the language pack provides. Creating a transpiler for languages to be vi-
Figure 4.4. An expression, its abstract syntax tree and the visualization with Jsvee showing the corresponding operations related to each node in the tree.

Visualized requires work but still, in this way it is possible to keep the core of Jsvee language-independent and then give developers the freedom of producing the operation sequence in any way they like to create it.

Step 2: Checking the Required Language Features

After the operation sequence exist (or while constructing it), the developer has to check that the language pack contains all the required language features and built-in functions. If something is missing, the language pack must be extended or the original example must be changed to take account the limitations of the supported features.

Typical cases with the Python language pack are that, for example, there should be a method for lists or strings which is currently missing. These new methods must be emulated with JavaScript and it is up to the developer how the required method is implemented. Optimally, this step can be skipped or if additions are required, there is a specific developer who maintains the language pack and extends it.

Step 3: Validating the Animation

When the animation should be ready for use, it must be carefully viewed to see that the animation behaves as expected. Because there is not a real

```plaintext
max(5, abs(2 * -3))
```
execution environment, the execution is purely based on the operation sequence and the functionality of the language pack. If there are errors in the sequence, the animation behaves unexpectedly.

**Step 4: Injecting the Animation to Animation Library**

Typically, all the animations are stored in a single JavaScript file which contains all the animations related to single course or single topic. This library is loaded together with the Jsvee core library. When a new animation should be created to the web page, the Jsvee core module finds the animation definitions and can construct the animation.

A ready animation definition is a JSON object which contain the operation sequence as a list of operations and a settings object which defines the layout for the animation and some other properties. For production use, the Jsvee core and all the Jsvee related JavaScript files can be minified to a single JavaScript file and the definitions to another file. In this way, there is a common library for the Jsvee and another file for the animations which can be easily changed based on the required set of animations.

### 4.5 Empirical Evaluation

In this section, we report the research we have done with Jsvee, how Jsvee has been used in CS1 courses in Finland and abroad, and what kind of results we have obtained.

#### 4.5.1 Using Jsvee in CS1 courses

The Jsvee library has been in use since Fall 2013. Program visualizations created with the Jsvee library have been used in multiple CS1 courses. For example, at Aalto University there are over 50 visualizations in Scala programming language and over 20 visualizations in Python integrated into CS1 course materials in two different courses.

Outside Aalto University the biggest user is the University of Pittsburgh. In their CS1 courses, there are over 50 program visualizations both in Java and Python. University of Jyväskylä has also integrated some of the existing visualizations and a few animations in C# in their CS1 course materials.

The number of students who have used the Jsvee animations such that we are aware of the usage, is over 2000. The library has been in real use
in courses which have hundreds of students each semester.

Because a working transpiler only exists for Python, the animations using Java, Scala and C# have required manual work to create the operation sequence. The output of Python transpiler can be used in some cases as a starting point, but also a prototype of another transpiler which parses block-based languages exists. The language it parses resembles C and Java but is not any real programming language. That transpiler is implemented with the Coco/R parser generator\(^3\).

### 4.5.2 Visualizations as Part of Online Materials

Publication III reports how students used Jsvee animations in a CS1 course at Aalto University. We offered over 50 program animations in Scala programming language embedded in the online course materials. The materials contained a kind of a self-study kit which consisted of theory, program animations, coding exercises and multiple choice questions. The course had only three lectures, and these online materials were the primary information source for the students.

The program visualizations were tightly integrated with the course materials and partially pedagogical decisions were made with the expectation that students view the animations. The animations had different roles in the materials. These are described in Publication III in Table 1. To summarize, most of the visualizations were aimed at clarifying newly introduced concepts. Other roles are, for example, clarifying example programs, predictive tracing or introducing new concepts via visualizations before describing them in the text.

In Publication III, we had three research questions. The first is related to the overall usage of the animations: do students view them although exercise points were not directly awarded? The second research question is how students use the animations: do they view the whole animation or only parts of them, do they ever move backward to see some steps again, or do they view the same animation multiple times? The third question asks how attentively students view the animations, and was researched by measuring how much time students spend in each animation step.

Our first result is that students really used the program animations. Figure 4.5 taken from Publication III shows the percentage of students who used the animations in each week. In the case of the novices, the

\[^3\text{http://www.ssw.uni-linz.ac.at/Coco/}\]
percentages range between 81% and 98%. For more experienced programmers, the percentages are lower, but that is expected as the visualizations are especially aimed at the novices.

Our results in PV usage are more positive than Alvarado et al. [5] reported. In their experiment, they had a similar ebook with text and program visualizations embedded into the materials, but they reported that already in the third week only 36% of the students used the program visualizations anymore, and after that it drops below 20%. There can be many factors which affect the number of students who actively use the visualizations, such as how well the visualizations are part of the text and how the materials motivate students to use them. Of course, the way of visualizing the example programs can have an effect.

The second result in Publication III is that students tend to view the whole program animation if they start viewing it. However, it is unlikely that students view the animation again in another time or move backward. These are similar results that Alvarado et al. obtained. We noticed that the finish rate correlates with the number of steps. The finish rate drops below 90% if there are more than 50 steps. If there are about 30 steps, the finish rate is on average 95%.

The third result shows that the students view the animations with relatively high speed. On average, they spent about 0.7–1.7 seconds per step depending on the visualization. The speed increases towards the end of
the course. We analyzed if students spent more time in some steps than others. We noticed that there are peaks especially in steps which are visually complex in Jsvee.

A peak in this method we used occurs when a student spends more time than average on a particular step. We analyzed these peaks collectively by counting how many students in total had a personal peak at each step. This revealed that there were many cases in which we would have liked to see a peak, but it did not exist. This suggested that it is likely that some part of the students may have missed relevant steps. This observation was one of the reasons why we started to think how to indicate the important parts of the animations and the process of implementing Kelmu began.

By using the Python additions to Jsvee, a CS1 teacher has been able to extend her course materials with new interactive program visualizations. The Python transpiler has helped a lot in the process because it is possible to try out different kinds of animations and see which is technically possible and also suitable for the materials when considering the pedagogical influence and the learning objectives.

4.5.3 Visualizations as Voluntary Practice Content

Publication IV reports another research setting in which Jsvee animations were used in a CS1 course at the University of Pittsburgh and Winston-Salem State University.

In this setting, annotated examples, program animations called as animated examples, and code execution problems for self-assessment were offered to the students. Annotated examples are code listings in which there is a toggleable textual explanation for each code line but the execution is not visualized in any way. The main motivation for this research setting was to find out how animated and annotated examples affect learning and are there some differences in engagement, for example.

Exercises were voluntary practice content for the course. In the first run, there were 31 program visualizations and in the second run 52. All the content was available through Mastery Grids portal [62]. There was a pretest before the course and a postest after the course to measure the knowledge level of students.

The results showed that students spent in total more time with the annotated examples and viewed more of them. This can be explained by the nature of the annotated examples: there are only a few explanations
to toggle instead of viewing a longer animation. However, students were more likely to complete the program animations they viewed, rather than the annotated examples. This result may indicate that interactive animations are more engaging than code examples which have only static code.

Another result is that there is a marginally significant positive correlation between the post-test results and the number of viewed program animations. Also the number of viewed animated examples and course grade had a marginally significant positive correlation. In addition to being engaging, these results show at least some marginal evidence that program animations can enhance learning.

In five-point Likert scale to describe the usefulness of the program animations after the course, students gave on average the result 3.44. Those students who had a low pretest result gave more positive answers and students with very high pretest results found program animations to be less useful. These results validates our assumption that animations are more useful for novices. We can consider it as a positive result that the average is above three; the usefulness is more positive than negative.
5. Summary

The research question for this part was formulated as follows:

**RQ1** How can we help **developers** to create interactive program animations for multiple programming languages?

a) What kinds of features are needed?

b) How can we design such a toolkit?

We have presented some of the existing program visualization tools which have the typical feature set in the current tools. Based on the tools, we have recognized those basic features which a program animation library should offer. These are the graphical representation, core functionalities to control the animation, and support for the concepts taught in typical CS1 courses. In addition to these, the library which would act as the building blocks for creating program animations should support expression-level visualization to provide better understanding of notional machines and a way to implement programming language specific features. All of these should be combined in a library which can create program animations for webpages as that is a platform-independent solution and can take advantage of modern web technologies.

As explained with Jsvee, we can design an extensible library which provides a set of ready-made operations to manipulate the notional machine the library implements and can visualize. The library can be extended with language packs which are responsible for evaluating the language-specific expressions which contain operators and built-in functions. The language packs can also add new language-specific operations with new animations to cover those concepts which are not part of the intersection of the features typical programming languages offer.

We have used Jsvee to create program animations in Python, Java, Scala, and in C# programming languages and these over 150 animations.
Summary

have been part of CS1 courses in Finland and abroad. Our results indicate that students find the animations useful because they use them without getting any exercise points. We have recognized that multiple factors such as the availability of other materials, the level of integration, background knowledge of the learners etc. are factors which can affect the results of using program animations. Therefore, we cannot say that learners would always use program animations made with Jsvee as much as they did in our experiment, but we have evidence that the strong integration between program animations and other materials can encourage novices to use program animations and give a concrete way of thinking about the notional machine.
Part II

Distributing Online Learning Activities

The second part of this thesis is related to distributing online learning activities and what kinds of problems arise in the process of making the content available online. Interoperability issues are the major challenge which we describe in more detail. To make the distribution process easier especially for DEVELOPERS, we present Acos server which hosts online learning content and can provide activities for multiple online learning management systems that may use different interoperability protocols. We give details of the architecture, how the server has been used to host content for programming courses, and a short evaluation of the suggested solution.

Relevant publications for this part are Publications V and VI. The research question for this part is:

**RQ2** How can we help DEVELOPERS to distribute online learning activities and to reuse content in multiple learning management systems?

Source code and documentation for Acos is available at:
https://github.com/acos-server/acos-server
6. Problems in Distributing Activities

This chapter describes the current problems in distributing online learning activities. It is based on Publications V and VI.

6.1 Interoperability and Reusability

We discuss here the two main problems, interoperability and reusability, together because they define almost the same problem but the viewpoint is a bit different. Interoperability issues occur when there are online activities, which support only a certain communication protocol or protocols which prevent using the same content with multiple learning management systems (LMS). Reusability of the content means that the same content can be used in multiple LMSs. Some content may be reusable so that the same content can be used in those LMSs which support a specific communication protocol but not with others. Making them interoperable will remove this limitation.

The survey in Publication V asked instructors what kind of issues they face when adopting and using online learning activities. The survey was sent to the SIGCSE mailing list, and 50 persons answered the survey. The results revealed that one of the biggest problems is related to integrating online learning activities to the existing course materials (13 instructors) or learning management systems in use (20 instructors).

Interoperability means in the context of the thesis that LMS is able to launch external activities and provide some information about the learner, such as student id. The launched activity can send back the grade and other collected data such as a trace of how the content has been used. The problem is not relevant if the communication between the LMS and content is not required.

The major challenge of making online activities interoperable and reusable
is that there are different communication protocols to describe somewhat the same information. For example, proprietary LMSs which the universities have developed often use their own protocols to communicate with the content which is also developed in the same university.

To overcome these issues, more standardized interoperability protocols have been designed. Nowadays, the IMS LTI protocol [46] is supported by the major LMSs such as Moodle, edX, and Blackboard. However, implementing support for a standardized protocol solves the problem only partially. For example, the LTI protocol supports launching activities but only sending a limited amount of data back to the LMS. Besides, there are LMSs which do not support the LTI protocol and the content cannot be embedded into those systems. Implementing the protocols can be difficult because, for example, the LTI messages are signed with a shared secret and the messages contain a lot of information which must be stored in the client-side to submit the grade back to the LMS.

Another solution to make the content reusable and interoperable is to use adapters. Alario-Hoyos et al. have developed the GLUE! architecture [4] to integrate various content to LMSs. In their architecture, both sides, the LMS and the interactive content, have an adapter. These adapters convert the messages so that the connection and data transfer can be established. Between the adapters, there is a separate GLUElet Manager to coordinate the process. In this way, LMSs using different protocols can use the same content if there is an adapter for each LMS. The weakness in this architecture is that both the content and the LMS must be aware of the adapters and be able to use them correctly.

6.2 Extensibility

Another major problem in the survey in Publication V INSTRUCTORS pointed out is that they cannot customize the activities for their local needs. This problem is reported by 40% of the INSTRUCTORS. This issue is also raised in Publication VI. Typically, the online learning activities are not customizable, and they may use different terms or have otherwise different pedagogical approach. In this case, INSTRUCTORS must think how much additional value the content produces if it is not fully suitable and may even cause confusion or other problems with LEARNERS.

Extensibility would allow customizing the content for a particular need as well as potentially providing means to create new similar content with
the same tools which were used to create the existing activities. There should be a way to let INSTRUCTORS create new content based on their needs.

### 6.3 Discoverability

The third major issue reported in the survey of Publication V is the difficulty of finding external online activities to use. This is reported by 44% of the INSTRUCTORS. There are a number of sites which may have some visualizations, quizzes or other tasks but these are scattered over various personal or institutional web pages. Searching for suitable content is hard for INSTRUCTORS because there are no central repositories or catalogs to explore. Using Google or other means to find resources is time-consuming and does not guarantee that the best resources will be found.

Having a community which produces certain kinds of online materials would be a valuable source for course materials. This idea is not novel as there are already this kind of communities, such as the Greenroom community [31] which produces and delivers content for the Greenfoot environment.

### 6.4 Scalability

Scalability is one of the recognized problems in Publication VI. If there is a server hosting online activities, it should be able to serve a high number of requests without too much latency. Especially in the era of MOOCs, the number of students can be in thousands or even in hundreds of thousands and the infrastructure hosting the content must be able to handle the load. Waiting for the content to load reduces the usability, and it may weaken LEARNERS’ attitude to use the material.

Scalability can also contain issues such as how to increase the capacity of hosting the content under heavy traffic as well as how the same server infrastructure can host a larger number of activities.

### 6.5 Logging

As the interactive content can trace and produce high amounts of data about how LEARNERS use the content, there must be a way to collect this
Problems in Distributing Activities

data which can be used in educational data mining, for example. The xAPI [3] protocol, also known as the Tin Can API, is one well-known standard to collect interaction data. It is based on so-called learning record storages (LRS) which receive the messages and store them. All the messages in xAPI have the same actor-verb-object format, for example, Learner A solved exercise X.

This is one way to approach the problem of how to collect data. One of the problems is the message format that may not always be suitable because there can be raw data which is hard to transform into other formats. LRS must also be implemented and be available. Berg et al. [14] describe that one of the problems is the lack of consistency of what kind of data should be collected and how to make it usable across institutional boundaries. Even large amounts of data can be useless if the format is not described, it does not contain relevant information, or data sources generate data in different and incompatible formats.
7. Acos Server

This chapter describes the second technical contribution of this thesis, Acos server. It can host and share the same online learning activities to be used in multiple learning management systems by separating the content and communication protocols. The learning activities Acos hosts do not have to know which interoperability protocol is in use because Acos provides a uniform interface for the communication, regardless of the interoperability protocols used to communicate with the learning management systems (LMS). It also provides a way to create and share content packages as well as store relevant log data for educational data mining. This chapter is based on Publication VI.

7.1 Design Goals

Acos is designed to answer the previously mentioned issues in Chapter 6 related to distributing online learning activities. The server implements the idea of a lightweight proxy that could handle the communication required for interoperability as represented in Publication V.

Based on the survey in Publication V and our experiences in using and developing online learning activities, we chose interoperability and reusability as the major goals in the Acos architecture. The design rule for the architecture is that all activities must be usable with all the interoperability protocols Acos supports but without any additional work from Developers. Acos does not provide any new interoperability protocols but implements some of the existing ones and allows Developers to use them indirectly via the protocol-independent interface of Acos. To achieve the primary goals, the architectural design heavily separates the interoperability protocols (such as LTI [46]) and the content.

To describe the need and aims for Acos, let us first see the typical ap-
Acos Server

proach of using online learning activities without Acos as presented in Figure 7.1 below.

Figure 7.1. LMSs using different types of interactive content without Acos. Each content type must implement the communication protocol to communicate with the LMS. If there are different protocols, LMSs can use only those content types which support the same protocol as the LMS.

Each content type separately implements some communication protocol that LMS uses to communicate with the content. It can be the LTI protocol or any other protocol, but the problem is that the protocol is separately implemented for each type of content. There can be activities which use different protocols, and only those LMSs which support that particular protocol can use the content. Or it is possible that the activity supports two different protocols because both of them are implemented in the activity. This implies that developing interactive content also requires effort from DEVELOPERS to implement and understand these interoperability protocols in addition to developing the content itself.

With this traditional approach, the connection between the content and the communication protocol is very tight. The main goal of Acos is to separate the content and the interoperability protocols so that it is irrelevant for DEVELOPERS which interoperability protocols are used, and the same content can be easily used with multiple protocols without any additional development effort. Figure 7.2 illustrates how Acos accomplishes the goal of separating the content and communication.

As suggested in Publication VI, Acos is a kind of transparent proxy. The central idea is that Acos implements communication protocols to support existing interoperability protocols. LMSs communicate with Acos by using their native interoperability protocols without any modifications to the LMS. Acos translates the messages and provides a uniform interface for the content regardless of the interoperability protocol.

A DEVELOPER can implement the content so that it can communicate with the simple interface Acos provides. The DEVELOPER does not even have to know which protocol Acos uses to communicate with the LMS.
Acos Server

Figure 7.2. The design concepts of Acos server derived from Publication VI. The figure illustrates the main concept of providing a uniform and simple interface for the learning activities regardless of the actual communication protocol between Acos and the learning management system.

Acos takes care of using the interoperability protocol in the correct format. In addition to supporting multiple interoperability protocols, the same server instance can host any number of different types of content, as well.

This approach has two major benefits. First, developers do not have to worry about getting familiar with different interoperability protocols because all the communication occurs through the uniform and protocol-independent interface which Acos provides. By using the uniform communication interface, the activities can take advantage of those interoperability protocols Acos support without using them directly. Second, all the activities can be used with any supported protocol. By developing a new protocol, all the activities can be instantly used with the new protocol without any modifications to the activities or to the LMS.

7.2 Technical Implementation

This section gives a brief overview of the technical implementation of Acos server. For more details, see Publication VI.

7.2.1 Modular Architecture

Acos is designed to be modular. It has a lightweight core to run the server, and additional packages are installed to provide all the contents and support for the interoperability protocols. Figure 7.3 illustrates the architecture and modules.

The purpose of each type of package is described in below:

**Protocol packages** implement existing interoperability protocols (such as LTI) to enable communication between an LMS and Acos. This communication
Figure 7.3. The internal structure of Acos server. The three main components are protocols, content types, and content packages. The actual activities are inside the content packages, shown as yellow symbols. Content types provide the common libraries which the content packages need to provide the activities. Acos can also host tools, such as tools for creating new content.

consists of at least launching an activity and sharing information about the LEARNER but typically also sending at least the grade back to the LMS after the LEARNER has finished the activity. A protocol package can also define how the possible log data generated by the activities is handled and stored.

**Content types** provide the common parts which all the activities of the same activity type require. For example, in case of Jsvee, the content type package provides the Jsvee core library and stylesheets which all the Jsvee animations share.

**Content packages** contain the actual activities. Content packages use the common libraries the content type package provide and use the libraries to implement the actual content. For instance, Jsvee content packages contain the operation sequence for each animation the package contains. Content packages can be grouped by themes, programming languages, difficulty, etc.

**Tools** are special packages which the server can host. They mainly provide tools for INSTRUCTORS to customize or even create new content.
7.2.2 Technologies in Use

Acos is built by using the Node.js framework\(^1\). Node.js provides an efficient non-blocking and event-driven framework to build web applications with high throughput. On top of Node.js, we use the Express framework\(^2\) to make the web development easier.

We use Node Package Manager (npm\(^3\)) to distribute the Acos packages. All the Acos packages are standard npm packages which can be easily installed in the server environment, and the packages will automatically install all the required libraries and other modules.

In this way, we can use all the features Node Package Manager offers and we do not have to implement another way to deliver the Acos packages. As the package repository in npm is public and freely available, anyone can produce new Acos packages and others can install them from the repository.

7.2.3 Serving Online Learning Activities

This section describes how Acos technically produces the requested activity. For more details and code examples, see Publication VI. To follow the description, see Figure 7.4.

![Figure 7.4](http://acos/lti/jsvee/jsvee-python/hello_world)

**Figure 7.4.** The process of fetching content from Acos as described in Section 7.2.3. The figure illustrates how the different package types participate in producing the requested content. The parts in URL and the corresponding packages are emphasized with colors.

Launching and requesting interactive content starts when a Learner uses a LMS and navigates to a page which contains an external activity

\(^1\)https://nodejs.org/
\(^2\)http://expressjs.com/
\(^3\)https://www.npmjs.com/
that is fetched from Acos. The LMS typically renders an HTML page for the activity which has an iframe element for the content. In this case, the content for the iframe is loaded from Acos. These are the Steps 1 and 2 marked in the figure.

Depending on the protocol the LMS uses, the iframe may use different mechanisms to send the request to Acos. For example, in the case of the LTI protocol, it uses HTTP POST to fetch the activity. However, all the mechanisms use the same URL pattern for the request. The request URL contains four parts which are also highlighted in the figure: the protocol, content type, content package and the name of the activity. These four attributes explicitly define which activity is requested and which protocol will be used.

When Acos receives the request to launch the activity, it first extracts the previously mentioned four parts from the URL and searches the corresponding Acos packages to handle the request. During the next steps, the HTML response is constructed stepwise by the packages.

In Step 3, the protocol package first receives the request. It will extract any protocol-specific data from the request, such as the user id and the URL to submit the grade. The protocol package will generate markup to store this data in the response as the server is stateless. It will also add a protocol-specific client-side library to enable the communication between the activity and Acos.

In Step 4, the content type package receives the request. It will add the markup which is common for all activities of that content type. These include CSS stylesheets, common JavaScript libraries, etc.

In Step 5, the content package receives the request. It will render the markup which is required to construct the requested activity. Depending on the activity, the content package can render the markup for the whole exercise, or it can just render a container element with some attributes which the JavaScript-library producing the content reads and produces the activity in the client-side.

After Steps 3–5 are completed, Acos has produced an HTML page which contains the requested activity. All of the three main components produced the markup they require. In Step 6, the content will be transferred as a response to the request, and the activity will appear in the iframe element. In Step 7, the Learner will see the activity in the browser. The Learner is not aware of Acos because the server itself is invisible for Learners and provides only the requested content.
### 7.2.4 Two-Way Communication

Launching an activity by using various interoperability protocols is one achievement but Acos provides more. After launching the activity, the activity can send events back to Acos. There are two main purposes for this: sending the grade back to the LMS and collecting data about how learners use the content.

The protocol package injects in Step 3 a client-side JavaScript library to the response when the activity is launched. This library provides a uniform interface also for the client-side to send events back to Acos. Whenever an activity wants to send an event to Acos, it calls the provided JavaScript function and gives the event name and payload. After calling the function, the protocol code automatically adds protocol-specific details to the request and sends the event to Acos. Therefore, sending events is made easy for developers who can send JSON objects to the server always in the same way.

As in the launching phase, all the three package types (protocol, content type, and content package) can process the received event. For example, the content type package can handle some content-specific event types and the protocol package will send the grading event from Acos to the LMS.

In addition to the predefined grading event, another specially handled event is the logging event. If the client-side code wants to send logging events with arbitrary JSON data, Acos can persist them. This requires that the activity is launched with a specific logging key which groups all the logging events with the same logging key to the same log store. By using the secret counterpart of the logging key, the stored log events can be later fetched from the server. This mechanism provides a way to easily collect any relevant data for learning analytics and educational data mining. In addition to this persisting method, the protocol can also define a way for these events to be passed to an external server, if needed and supported.

### 7.3 Outcomes

We highlighted five problems in developing and using online learning activities. This section will go through the problems and discuss how Acos solves these issues.
7.3.1 Interoperability and Reusability

The main goal of Acos was to solve problems in interoperability and reusability. The result is the fundamental concept in the Acos architecture which separates the content and the communication protocols. When a developer develops content to be hosted in Acos, the developer does not have to know which interoperability protocol will be used. For the communication, Acos provides a uniform interface which is easy to use. For example, sending the grade from the client-side code always works in the same way regardless of the protocol used between Acos and the LMS. It is responsibility of the Acos protocol package to handle the event and implement the actual communication between Acos and the LMS. The developer who implements new content does not have to be aware of any details about how the protocol package works.

This separation between the content and communication also helps with reusability. As the content does not have any protocol-specific parts, it can all be used with any communication protocols Acos supports. To use the content with a new LMS which uses an unsupported interoperability protocol, only a new communication protocol package in Acos must be implemented, and all the content will be instantly available without any modifications.

7.3.2 Extensibility

To address the issues with extensibility, the Acos architecture is modular and can be extended by adding new packages to the server. To create new content with this architecture, it means that one of the existing content packages is extended with new activities, or a new content package can be created. The separation between the content type and content package divides the common parts and the activity-specific parts to separate packages. Therefore, a new content package implements only the activity-related parts and uses the common libraries and other features the content type package provides.

Tool packages can provide means to create new content easily for local needs, or as in the case of Kelmu, provide an editor to tailor the existing Jsvee animations, for example.
7.3.3 Discoverability

In addition to seeing Acos as a way to use the same content with multiple interoperability protocols, it can also be seen as a repository of activities. Acos can host many types of content, and each content type can contain an unlimited number of content packages. An instructor can visit Acos and try all the activities in the browser and see which activities could be useful. After choosing the content, the instructor takes the URLs for activities and inserts them to the LMS.

Acos packages also contain metadata about the content. To access the content directly from an LMS, Acos provides a RESTful API which can be used to retrieve a full content listing, or query the available activities by keywords, for example. This approach would allow a tight integration with an LMS so that the instructor can search and select the content directly in the LMS without visiting or even knowing anything about Acos.

7.3.4 Scalability

One of the reasons for using Node.js is closely related to the requirement of scalability. Node.js provides a non-blocking and event-driven framework which can serve a very high number of concurrent requests. Another major architectural decision is to make Acos stateless. After an activity is launched, Acos server-side components does not know anything about the launched activity. This is achieved so that the protocol packages store all the relevant session information to the HTML response instead of using server-side sessions.

If there is too much traffic for a single Acos server, it is easy to create more servers and use them together with a load balancer because the server is stateless. Instead of using a central activity repository, it is also possible to set up a local Acos server and install a relevant subset of the packages to that server.

7.4 Acos in Production Use and User Evaluation

Aalto University runs an instance of Acos which can be freely used to access the contents it provides. Currently, Acos has three main content types to be used in CS1 courses: Jsvee program animations, Js-Parsons code construction exercises, and so-called combo exercises which combine
Js-Parsons and Jsvee to give a visualization of the constructed program. In addition, there is a prototype implementation of creating content packages for JSAV algorithm visualizations.

We know that at least 1300 students from six universities (Aalto University (Finland), Arizona State University (USA), University of Jyväskylä (Finland), Tampere University of Technology (Finland), Universidade Federal de Uberlândia (Brazil), and University of Pittsburgh (USA)) have used the server and the activities it hosts.

As described in Publication VI, we gathered feedback from those Instructors who have used the CS-related activities Acos currently provides. We have omitted Learners because the server is invisible to them. Although the number of students who have used the activities in Acos is quite high, the number of Instructors is low, and therefore these opinions should be treated as initial comments.

Instructors are happy of the ease of integrating the content to their LMSs. Also, the latency of using the shared server is low although the content is hosted in Finland and used abroad, such as in the United States. However, one of the Instructors highlighted that although the technical integration is easy, the concept of using shared activities possibly designed with a different pedagogical approach can be problematic. We discuss the challenge of different pedagogical contexts more in Part III.

Developers’ point of view is that the usability of the server is important, and currently implementing the first running version of new content takes only a minimal amount of time. Using the Node.js framework and its package manager also simplifies the content creation. There is some critique that the separation between the content types and content packages might be better, and it should be easier to create new content packages with less coding. However, the technical documentation in GitHub was considered helpful.

One of the Instructors pointed out the importance of the logging features although they were not initially part of the specification and implementation. The current way of logging has been proven to provide valuable datasets for educational data mining.
8. Summary

**RQ2** How can we help DEVELOPERS to distribute online learning activities and to reuse content in multiple learning management systems?

In Part II, we have addressed the problems in distributing of interactive learning materials as well as making them reusable. Based on the questionnaire in Publication V, we know that finding and integrating online activities are the major problems which limits the usage of online activities. Therefore, we have presented the Acos server which can host online learning activities, and the server offers a way to create content packages which can be easily distributed. The server can be seen as a concept repository which INSTRUCTORS can browse and select content for their needs.

The major outcome in Acos is that it separates the content and the required communication between an LMS and the content. Acos implements a few existing interoperability protocols such as the LTI protocol and offers a simple interface for DEVELOPERS to enable the two-way communication. This interface is always the same regardless of the actual interoperability protocol in use. By implementing this interface, content developers can use the same content with multiple interoperability protocols without any changes to the content. In addition, Acos provides a mechanism to collect any log data DEVELOPERS would like to collect about the usage of the activities.

Acos has been used in multiple CS1 courses to provide interactive materials. There are not yet many developers but the feedback we have collected so far indicates that the server architecture seems reasonable and provides a quick way to integrate new content types to existing LMSs.
Part III

Tailoring Animations

The third part of this thesis focuses on how INSTRUCTORS should be able to tailor automatically generated and generic program animations and other interactive animations to a particular pedagogical context. We give an overview of theories which explain the learning process especially in multimedia learning context and how it is possible to enhance animations in certain ways to support learning. The main contribution of Part III is the Kelmu\(^1\) toolkit which allows augmenting animations with arrows and textual explanations. We describe the technical implementation of Kelmu, how the implementation is justified by various theories, how the toolkit has been used, and what kind of preliminary results we have obtained.

Relevant publications for this part are Publications II and VII. The research question for this part is:

**RQ3** How can we help INSTRUCTORS tailor generic interactive animations to particular pedagogical contexts?

Source code and documentation for Kelmu is available at:
https://github.com/Aalto-LeTech/kelmu

\(^1\)Kelmu means a transparent layer in Finnish.
9. Related Work

9.1 Cognitive Load Theory

To understand how the human cognitive architecture affects learning, let us discuss the cognitive load theory (CLT), introduced for the first time in literature in 1988 by John Sweller [112]. The assumption in the cognitive load theory is that human working memory is limited and can hold only a few active elements at the same time. The theory describes the limitations and mechanisms of the human brain to learn new concepts and how the learning process can be made more effective by forming the study materials so that they take advantage of the theory to reduce cognitive load.

There are a few variations of CLT, but the most-widely accepted version of cognitive load theory as presented by Moreno and Park [83] consists of three components that form the cognitive load: intrinsic load, extraneous load, and germane load. These load factors should fit in the total working memory capacity. Figure 9.1 represents the total working memory capacity and the different factors of cognitive load. We introduce the three factors below as Moreno and Park have described them.

**Intrinsic load** is based on the difficulty of the concept the learner is studying. There is always at least some number of elements that must be held in the working memory when working with the problem and therefore intrinsic load can not be reduced below a certain limit that is based purely on the concept or problem. For example, summing up two numbers require that the numbers must be in the working memory.

**Extraneous load** is caused by how the learning materials are presented
According to the cognitive load theory as presented by Moreno and Park [83], the total cognitive load in working memory consists of three different elements: intrinsic, extraneous, and germane load.

and the load should be minimized. CLT aims at reducing extraneous load by providing some guidelines to follow when designing materials.

**Germane load** covers the load the learning process causes. Scheme acquisition and automating tasks produce germane load and therefore learning becomes better if the share of the germane load can be larger by providing interesting and motivating materials. However, all these three sources of the cognitive load must fit into working memory at the same time.

Although this traditional and widely used model of cognitive load theory consists of these three presented parts, some recent research (e.g., Schnotz [100], Sweller [114]) questions the germane load as a similar distinct source of cognitive load as intrinsic and extraneous load.

### 9.1.1 Storing and Retrieving Information

Sweller's article about the recent theoretical advances of CLT [113], which is the main source for this section, describes the *long-term memory* as the primary driver and the central structure of human cognition. It is also the main component in problem-solving and thought.

To explain how the information is acquired, Sweller discusses the *borrowing and reorganizing principle*. It states that most of the information in long-term memory is obtained by imitating other people, listening to them, or reading. However, the exact wording of what somebody said, for example, is not stored but humans construct a representation which extends the current knowledge.
**Related Work**

*Schema theory* reflects the process of generating and storing knowledge. A schema is a model of information which is constructed from multiple information sources but can be treated as a single element in long-term memory. Learning can be seen as constructing the existing schemas to make them available for problem-solving. Automating the schemas make problem solving faster because the process does not have to be consciously considered. However, modifications to long-term memory should not be too large. They should be incremental and slow.

To solve problems humans use this gathered knowledge. Still, in many cases, there are multiple alternative ways to continue with the problem but no previous knowledge about which option leads to the best solution. In this kind of situation, *random generation* is needed to test the effectiveness of the different solutions by trying them out. The information received from these tests will create new knowledge which will be later applied to solving similar problems.

Using course materials is one important source of constructing and altering existing schemas. Therefore, it is important that the materials link concepts between others to create connections between existing schemas. Also, there should not be too much new information at once because this weakens the information storage in long-term memory.

### 9.2 Multimedia Learning

Richard E. Mayer discusses in his book *Multimedia Learning* [71] how to design effective multimedia presentations for learning and defines twelve principles that should be taken into account. Theory of multimedia learning suites well to this part of the thesis because it extends the cognitive load theory and works well with the concept. Moreover, it gives concrete guidelines on how to create effective multimedia-based learning materials for learners.

Based on the book, we first present the idea of multimedia learning, give a short description of the principles and then focus on a few relevant principles for this thesis.

#### 9.2.1 Definitions and Use of Multimedia in Learning

Mayer's definition of *multimedia instruction* is “a presentation of material using both words and pictures” [71, p. 5]. Words mean in this context
that the material is presented in a verbal form, such as printed or spoken text. Pictures can be static graphics, graphs, photos, maps, or animation or video. Multimedia learning is learning from words and pictures.

Mayer constructs the idea of multimedia learning by using an assumption that humans can process information in two ways: there are systems for verbal and visual material. Presenting information both verbally and visually takes advantage of humans’ capability to simultaneously process these sources. However, Mayer does not assume that words and pictures are equivalent ways of presenting the same material. Some information is received better when represented as pictures, some as words. But still, learners are able to create a deeper understanding while building connections between words and pictures than from using them alone.

9.2.2 Metaphors of Multimedia Learning

Mayer [71, p. 14–19] provides three contrasting views for multimedia learning: multimedia learning as response strengthening, multimedia learning as information acquisition, and multimedia learning as knowledge construction.

Response strengthening in the context of multimedia learning means the way of strengthening correct information and similarly weakening false assumptions. For example, in a quiz, the game can provide graphical feedback to indicate the correctness of the answer. This feedback can be seen as a reward or a punishment to reinforce a correct answer. It is the instructors’ responsibility to design the responses so that they fulfill the learning goals.

In the information acquisition view, the goal of multimedia is to deliver information. A learner is a passive being who tries to store the presented information in memory. Mayer uses a metaphor in which the teacher pours information in the learner’s mind, but this is not possible as humans cannot store an unlimited number of exact definitions or other new concepts.

Multimedia learning as knowledge construction emphasizes the view in which a learner builds a coherent mental model from the presentation. Information cannot be “poured” in someone’s head, and therefore, the learner must have an active role because knowledge is personally constructed. Because of this, the same presentation shown to two learners can result in two different learning outcomes. The instructors’ role is to assist the learners and act as a cognitive guide. Mayer favors this
approach because it is more consistent with the research on how people learn. Mayer states:

“I see the goal of multimedia as helping people develop an understanding of important aspects of the presented material.”

### 9.2.3 Multimedia Learning Outcomes

Mayer [71, p. 19–21] presents two major goals for learning: remembering and understanding. Remembering means that a learner is able to reproduce or recognize the presented materials. Understanding is deeper learning and can be achieved if the learner has been able to construct a coherent mental representation from the material. The learner is also able to apply the learned concepts and answer questions that cannot be directly answered based on the presentation.

To define the level of learning, Mayer also presents three kinds of multimedia learning outcomes: no learning, rote learning, and meaningful learning. At the lowest level, no learning, the learner is not able to remember almost anything from the presentation and cannot answer questions that require only remembering. Rote learning is defined so that retention is good but transfer is poor. The learner is able to remember at least some parts but not necessarily understand everything and cannot use this knowledge in practice. In meaningful learning, both retention and transfer are good. The aim of multimedia learning is to provide ways for meaningful learning.

### 9.2.4 Mayer’s Principles of Multimedia Learning

Mayer [71] presents twelve principles that should be taken into account when designing multimedia presentations.

1. **Coherence Principle:** People learn better when extraneous material (such as additional words and pictures) is excluded.

2. **Signaling Principle:** People learn better when essential parts of the material are highlighted.

3. **Redundancy Principle:** People learn better from animation and narration than from animation which contain the same information as narration and text.

4. **Spatial Contiguity Principle:** People learn better when corresponding
words and picture are near each other.

5. **Temporal Contiguity Principle:** People learn better when corresponding words and pictures are presented at the same time instead of presenting them successively.

6. **Segmenting Principle:** People learn better when the multimedia presentation is user-paced instead of a continuous presentation.

7. **Pre-training Principle:** People learn better when they are already familiar with the vocabulary used in the presentation.

8. **Modality Principle:** People learn better from graphics and narrations than from graphics and on-screen text.

9. **Multimedia Principle:** People learn better if there are also pictures and not only words.

10. **Personalization Principle:** People learn better if words are in conversational style rather than formal style.

11. **Voice Principle:** People learn better if the voice is natural and friendly instead of machine-generated narration.

12. **Image Principle:** People may not learn better if the image of the speaker is visible.

Mayer and Moreno [74] give more examples how to apply these to practical use. Four of these principles which are the most important to this thesis are discussed in the following sections.

### 9.2.5 Signaling Principle

"People learn better when cues that highlight the organization of the essential material are added." [71, p. 108]

In general, extraneous material should be avoided, but it is not always possible, as some background information or related concepts are often required. Therefore adding cues to highlight the essential parts of the material help learners to focus those parts more carefully. Giving these cues is called **signaling**.

Mayer [71, p. 108–117] describes various ways how the signaling principle can be applied to written text, audio, or other kinds of multimedia. In written or spoken text, outlining the content gives an overview of the key concepts. Similarly, using words such **first**, **second**, and **third** points out the main content. Also speaking keywords in a louder and slower voice...
can highlight those words better.

In the context of interactive animations, the visual signaling techniques are more interesting. Mayer gives five different ways of emphasizing the content visually:

1. **Arrows** can point to the integral parts of figures or other elements in an animation to draw learner's attention.

2. **Distinctive colors** can emphasize those parts which are important in red, for example, whereas the other parts of the illustration are drawn in black.

3. **Flashing elements** give a clear indication which part of the complex illustration is important at each step of the animation.

4. **Pointing gestures** by an onscreen agent can point out the specific location to look at.

5. **Graying out** the unnecessary parts. The part which is relevant can be shown in a “magnifying glass” and the rest of the picture is blurred or grayed out.

Mayer reports six studies in three articles [40, 66, 111] in which the effect of signaling principle was measured. In five studies, the transfer test produced positive results implying the signaling principle may be effective. The multimedia content in the experiments was about lightning, airplains, and biology. They were implemented as paper illustrations or interactive computer animations. The two experiments made with computers have the largest effect size. However, Mayer states that as the results are not that strong and based only on six experiments, the effect should be studied more.

### 9.2.6 Spatial Contiguity Principle

"Students learn better when corresponding words and pictures are presented near rather than far from each other on the page or screen." [71, p. 135]

By presenting corresponding words and pictures near each other, learners do not have to visually search the corresponding elements and are more likely able to hold them both in working memory at the same time. Cognitive resources can be used for active learning and build mental models by combining the words and the visual representation.

Mayer reports results from thirteen research settings in eight research
Related Work

articles [70, 75, 82, 115, 22, 118, 17, 49] in which spatial contiguity principle seems to have a positive effect on learning. The visualizations were about a broad variety of topics: brakes, lightings, mathematics, engineering, heart, tire pumps, statistics, and physics. Mayer reports that the effect seem to be larger in book-based material than computer-based material. One reason for this might be that in computer screen the elements are closer to each other than in traditional books.

9.2.7 Segmenting Principle

“People learn better when a multimedia message is presented in user-paced segments rather than as a continuous unit.” [71, p. 175]

Segmenting principle suggests splitting the content into smaller parts, and a learner can decide when to move from one part to another. Viewing a continuous multimedia presentation may overload working memory if the presentation moves too fast and the learner is not able to process the information and build or expand the current mental model.

Mayer justifies that giving the ability to control the pace and when to move to the following step, the learner can complete the deeper cognitive processes of organizing the received information. After this process is done and the learner is ready to move forward, the learner can advance the presentation, receive then next part of the information and continue processing that. In this way, the learner does not have to worry about missing important details as there is enough time to concentrate on each step.

Mayer gives results from nine experiments in six research articles [72, 73, 67, 56, 8, 35] that measured the effectiveness of segmenting the presentation. The visualizations were related to lightning, electrical motors, geography, chemistry, mathematics, and probability. In all cases, the learners who saw a segmented version of the presentation performed better on problem-solving transfer tests, which suggests segmenting principle can lead to deeper learning.

9.2.8 Modality Principle

“People learn more deeply from pictures and spoken words than from pictures and printed words.” [71, p. 200]

Students learn better when spoken text is used instead of printed text.
Related Work

The reason is that if pictures and words are presented visually, the visual channel becomes overloaded but the auditory channel remains unused. Therefore, using audio instead of visually presented text enables using both channels.

Mayer [71, p. 200–220] refers to information-delivery theory which states that multimedia learning is improved by using as many routes as possible to provide information. According to this theory, audio and pictures, as well as, text and pictures use two routes, and therefore they should have equal learning results. In Mayer's opinion, this theory is outdated because it does not use the cognitive theory of multimedia learning and its dual-channel assumption. It states that humans have two channels for receiving information: one for visual and one for verbal processing. Using these together balances the cognitive load and enhance learning.

To show the effect of modality principle, Mayer presents results from 36 experiments (see pages 210–213 in the book [71]) which measured the effect of the modality principle. Only in one case out of 36 experiments, the effect size was negative. This gives a strong assumption that the modality principle enhances learning.

However, modality principle is seldom applied because typically, all the information is presented as text and pictures. Producing high-quality audio for this purpose requires a lot of work and may be one reason why it has not become a more common way to provide information.

9.3 Integrating Multimedia Learning with Program Animation

In addition to Mayer's results, also Plass et al. [89] report similar principles to create efficient animations. They have a comprehensive set of references to research in this field. This brings up a question how we could take advantage of these principles in the context of program animations.

If we take a look at the represented visualization tools in Section 2.3, they do not give any hints about the important steps because animations are generic. A Learner should and must be aware of the learning targets and notice them because they are not signaled. Therefore, pointing out the most relevant parts of the animation would be useful, and Learners can especially concentrate on those steps.

However, signaling in program visualization tools is not novel, although it is rare. For example, in VIP system [122], Instructors can visualize C++ programs but also write comments about the execution. The feature
is implemented so that code comments with a specific format are recognized, and the description at the higher-level of abstraction is shown for **learners** when the execution reaches that particular point.

Among the program visualization tools described in Section 2.3, UUhis-tle and ViLLE can provide automatically generated explanations about the steps they visualize. However, the location for this information is in both systems in the lower-left corner of the window and far away from those parts of the user interface where most of the actions take place. This means the spatial contiguity principle is not very well followed.

Typically, program visualization tools support the segmenting principle. They provide a stepping button which proceeds the animation each time when the button is clicked. This gives time for the learner to think about the animation step. Also, a button to activate a continuous animation often exists although it can be considered harmful. If the animation speed is too fast, the **learner** does not have enough time to think about the concepts shown in the animation.

Mordechai Ben-Ari [12] raised the issue that the gap between the technical quality of program visualization systems and their lack of pedagogical support is hard to bridge. He suggested a tool to add pedagogically important additional features as a separate module which has a lightweight module for communication with the animation tool. The module could trace the animation execution and show explanatory texts at pre-defined steps.

An early prototype of such tool based on Ben-Ari’s idea was created by Traff [119]. The outcome is LOjel, a version of Jeliot 3 which has a new area on the left-hand side of the application for explanatory texts. These texts are manually created and configured to appear at particular steps to provide the pedagogical context.

The existing approaches in program visualization tools show that implementing this kind of features which follow the multimedia learning principles is possible. However, instead of implementing the features to each tool separately, we would like to find a more generic way to add similar features to any web-based dynamic animations. Our result for this, the Kelmu toolkit, is presented in the next chapter.
This chapter describes the Kelmu toolkit which instructors can use to tailor existing generic online animations to contain context-specific pedagogical annotations such as explanatory texts. Annotations help to follow Mayer’s multimedia learning principles [71], and the augmented animation should enhance the learning experience.

10.1 Overview of Features

To get familiar with Kelmu, we start with a short description of the features it provides. After this overview, we discuss in more detail the design goals and technical implementation of the toolkit. An illustration of Jsvee animation which is tailored with Kelmu is presented in Figure 10.1.

Figure 10.1. A Jsvee program animation which is augmented with Kelmu. Kelmu provides the red explanation text and the arrow on top of the original animation.

When an instructor tailors an existing animation, the typical workflow consists of adding arrows and textual explanations which signal the integral steps. These annotations are drawn on top of the original ani-
mation by using the WYSIWYG editor Kelmu provides. The red text and arrow in Figure 10.1 are added with Kelmu.

The animation can be augmented, in addition to the visual annotations, with audio and additional controls which can control the stepping of the animation. For example, an instructor can create a shortcut for those learners who are already familiar with a concept that is explained next, and by pressing this additional button, the animation moves multiple steps forward. The instructor is also able to always skip certain steps or show them as a single step. Because the automatically produced animations are typically generic, the code often contains lines which are not relevant in that particular pedagogical context, and can be skipped. This helps to reduce the total number of steps and maintain a reasonable length for the animation.

Kelmu supports a feature called substeps. By using substeps, an instructor can design a set of annotations which are shown sequentially before the animation continues. This can be thought as duplicating a particular animation step and having different annotations for the duplicates. This is useful if an animation step contains independent aspects to be explained separately, for example. In this way, the learner can concentrate on one aspect at a time. Substeps can also be used if the animation area is small and all the information does not fit the screen.

10.2 Design Goals

As explained in Section 2.3, automatically or semi-automatically generated animations are typically generic. For example, in the field of program visualization, an arbitrary code can be typed, and the tool produces the corresponding animation of the program execution. But because the tool which produces the animation does not understand the context in which the animation will be used, the result is context-independent and cannot emphasize those steps which are important in a particular use case. It is also possible that the same animation can be used in multiple contexts which should highlight different aspects of the animation.

Design goals, as also described in Publication VII, for the Kelmu toolkit are that it should support tailoring especially Jsvee animations but other animations as well. The main motivation for the toolkit is to provide for instructors means to add pedagogical context in a generic animation.

Kelmu enables instructors to use four principles of Mayer’s multime-
dia learning principles which were earlier presented in Sections 9.2.5–9.2.8. Signaling principle is probably the most important principle. Signaling helps learners to notice the important parts, but also allows instructors to provide the pedagogical context for the animation. Signaling is achieved by annotating the important steps with explanatory texts in distinctive colors such as red.

Spatial contiguity principle is related to the previous principle. When an instructor adds explanations and arrows, the instructor can drag and drop the annotations near to the exact location in which the action occurs. Placing the additional elements next to the position in which the explained operation takes place, the learner will more likely notice the action and read the explanation.

Segmenting principle is taken into account so that Kelmu actually requires that the animation must be advanced by clicking a button. A continuous animation without user-controllable stepping is not supported.

Modality principle is also supported by the Kelmu toolkit. Instructors can add audio clips for each step. The clips may contain spoken explanations or other sound effects to point out that something important happens.

### 10.3 Using Kelmu

Kelmu has two modes. The first mode generates the augmented view for learners. As a learner only sees the augmented version, the presence of Kelmu as a separate component is not visible for them. After the underlying animator is ready, Kelmu interacts between learner’s commands to control the animation and the animator. The technical implementation is explained in the following section.

The second mode is meant for instructors. If the Kelmu editor library is loaded together with the Kelmu core library, instructors can tailor animations while viewing them. Kelmu shows a small editor which contains the buttons to create and modify the annotations. An instructor can drag and drop all the elements to the correct places and modify the properties such as the color and font size. Figure 10.2 illustrates the editor view.
10.4 Technical Implementation

This section gives an overview of the implementation of Kelmu. More detailed description and the requirements for animators are presented in Publication VII. Kelmu works as a transparent proxy. The term transparent has two different meanings in this case. Kelmu is technically transparent so that the underlying animator does not have to know anything about the Kelmu. The second meaning is that Kelmu provides a transparent layer on top the animation which Kelmu uses to draw the annotations, so that a Learner is not aware of the technical detail that there is a separate library to augment an animation.

To achieve the technical proxy without modifying the code of the animator, the proxy is implemented by changing the original event handlers created by the animator. To follow the description, see Figure 10.3. In general, browsers do not allow changing any existing event handlers. However, Kelmu needs to be able to listen to when the learner advances the animation to augment the next step.

To overcome this issue, the jQuery library\(^1\) provides a mechanism to customize the added event listeners. Now when the Learner clicks the step button provided by the animator, Kelmu receives the user event first. Kelmu forwards the event to the animator but remembers that after the animator has processed the event, Kelmu must augment the next step. To make this work, the animator must use the jQuery library to attach the original event handlers.

\(^1\)https://jquery.com/
As Figure 10.3 shows, the original event is not always delivered to the animator. If there are substeps, Kelmu does not send the event to the animator until all the substeps have been stepped through.

Technically, the annotations are drawn in their own layer which is positioned on top of the original animation area. Annotations are normal HTML and SVG elements. They are drawn in a separate container element to minimize the interference with the animator. Because the positions of annotations are stored as coordinate pairs, Kelmu assumes that the elements always appear in the same position. It is also required that the animation proceeds deterministically because the annotations are attached to a certain step number.

Publication VII gives more details about how Kelmu and the animator can optionally communicate with each other. The basic functionality does not require any communication, but if the animator supports the message passing protocol Kelmu uses, Kelmu can send arbitrary commands to the animator which can be predefined or sent from the buttons added to the annotation layer. With these custom commands, the animator can, for instance, hide or show some elements of the animation or provide additional features. With the message passing mechanism, Kelmu can also send requests to skip a certain number of steps without visualizing them at all. The animator can send a message after the transition animation is over, and Kelmu can instantly draw new annotations instead of waiting the default transition time every time.
10.5 Results

This section contains preliminary results of using Kelmu in CS1 courses. More detailed information about the results is in Publication VII.

10.5.1 Using Kelmu Increases Time-On-Task

In Publication III, we pointed out that there are many cases in which something relevant occurs in the animation, but it seems that the students did not stop to think about what they saw. It was possible that they unintentionally skipped important steps. The data for that analysis was collected in Fall 2014 at Aalto University during a CS1 course which is aimed at CS major students.

For the next semester when the same course was organized again, i.e. Fall 2015, all of the over 50 Scala program visualizations made with Jsvee were augmented by the instructor of the course. We collected the same log data to compare how the augmented versions of the Jsvee animations had changed students’ behavior.

One major result is students spend more time with the augmented animations than the original unaugmented animations. The difference can be from a few seconds to up to 30 seconds per animation, which means the increase is notable. If we calculate how many percent the time usage has increased in each animation separately, and take the average of the percentages, novices have spent on average 29% more time with the augmented animations than the novices the year before without Kelmu annotations. Novices spent on average 1.3 seconds longer in each step if we calculate the total time spend in an animation and divide that with the number of steps. If we calculate the average time spent with each animation and take the average of those values, it is 87 seconds for the original animations and 103 seconds for the augmented animations.

To evaluate the cause of the increased time usage, we analyzed how the usage has changed when the animations contain annotations. We noticed that the major part of the additional time comes from those steps which are annotated. This can indicate that the signaling principle helps learners to notice aspects they might have earlier unintentionally skipped. Of course, there can be other explanations as well, and we cannot say based on this data how effective the increased time usage was for learning. However, we consider this preliminary result positive as one of our major concerns in Publication III was that students proceed too fast and are likely
to skip those concepts animations were trying to demonstrate.

10.5.2 Novices Read Annotations

In Fall 2015, in the same CS1 course as in the previous section, an experiment with an eye-tracking device was carried out. In this experiment, twenty CS1 students voluntarily participated the experiment. A very small number of the exercise points (less than one percent) were given to them as a reward. The participants did not have any previous programming experience. A RED250Mobile eye-tracking device made by SMI was used in this experiment.

During the experiment, each student viewed two Jsvee program animations. The first animation was related to function calls and the second to buffers and references. We assigned the students to four randomly selected groups. In each group, one of the animations was annotated, and the other was not. The difference between the groups was the animation they started with and which of the visualization was augmented with Kelmu.

Figure 10.4 shows an authentic example from the experiment. A heatmap of the areas which the novice viewed during a single step is visualized. On the left-hand side below the code area, there is a red annotation emphasizing the situation: there are variables with the same names but they have references to different buffers although the name of the variable is the same. In this case, no arrows were used to point the actual variables and the text is placed in an empty position.

Figure 10.4 clearly shows how the novice has read the explanation and then checked the corresponding parts of the visualization. In this experiment, 79% of the annotations in the first animation and 94% of the annotations in the second animation were read. These results give more evidence that the increased time usage is caused by reading the annotations, and if there are explanatory texts, students will very likely read them. However, we must remember that the number of participants was low and the controlled nature of the experiment can affect the results.

10.5.3 Learning Effects

The third experiment related to Kelmu was carried out in summer 2016. In another CS1 course which uses Python at Aalto University, there are program animations made with Jsvee, but Kelmu has not been used. This
course is offered for non-CS majors. In this experiment, we provided two additional program animations related to the basics of the function calls and references with lists. After viewing the animation, the visualization disappeared, and a static code listing appeared. We asked the students to predict what the given code will output when it will be executed.

Students were randomly assigned to three groups. One group saw only the Jsvee program animation without any additional explanations. The second group had quite a long textual explanation of the whole program code above the visualizations. The third group had the same information as the second group, but Kelmu was used to represent the information at particular steps. The groups were changed between the two program animations which were in two different exercise rounds.

The animations were in the learning management system like any other exercises, and the students could do the exercises during the exercise sessions or at home. The exercise had two parts: first, the student viewed the visualization normally and after that, the animation was removed and a small Python program became visible. The task was to read and interpret the code and write the output of the program. In addition to the answer, we also asked them to write a short description (max. five sentences) about what kind of logic they used to predict the result, and how certain
they were about the answer.

Students were not supposed to use an IDE or other aids to produce or check the answer, but as this was not a controlled experiment, we cannot guarantee that aspect. They were told that they would receive a few exercise points regardless of the correctness of their answer, so generally there was no reason to cheat.

The number of participants in this study was so small that our results are not statistically significant. In the first task related to function calls, the number of students who answered correctly is roughly the same varying from 42% to 60% between the groups. In the second task, those students who saw the annotated version of the animation about lists and references, 40% answered correctly to the code reading task, whereas of the others who did not see the annotations, only 10% and 18% answered correctly. But as the group sizes are so small (about 20 novice students per group), we cannot, unfortunately, say if this difference is caused by the annotated animation or is coincidental. This experiment should be repeated with larger groups.

10.6 Kelmu with Other Animators

Although the number of external DEVELOPERS or INSTRUCTORS who have used Kelmu to augment new or existing content is low, the feedback we have received is positive. For example, one instructor has adopted Kelmu and has used it to enhance some existing materials and create new animations which are not made with Jsvee. This shows that the architecture decisions of Kelmu are well founded and the idea of using it with animators other than Jsvee also works in practice. We have tested that Kelmu works well with Online Python Tutor and JSAV algorithm animations if someone would like to augment animations made with those tools.
11. Summary

**RQ3** How can we help **INSTRUCTORS** tailor generic interactive animations to particular pedagogical contexts?

Based on the cognitive load theory and Mayer’s multimedia learning principles [71], these theories give evidence that minimizing the unnecessary details and emphasizing the important parts of learning materials enhance learning. Therefore, we presented in Part III the Kelmu toolkit which **INSTRUCTORS** can use to highlight the key parts of interactive animations.

Because automatically or semi-automatically produced program animations are generic, **INSTRUCTORS** can use Kelmu to add explanatory texts that explain especially those key concepts which are relevant in that particular pedagogical use case and context. In this way, the same generic animations can be used in different pedagogical contexts depending on the annotations that have been added. Tailoring can also customize properties such as stepping so that multiple steps which are not relevant in that particular context can be skipped or shown as a single step to reduce the number of steps and increase attention in the most important steps.

We have implemented the Kelmu toolkit which can draw annotations on top of the existing animations and control the stepping. For **INSTRUCTORS** it provides a simple editor with WYSIWYG interface to create and edit annotations. While viewing an animation, an **INSTRUCTOR** can drag and drop the annotations to correct positions. **LEARNERS** see only the augmented version of the original animation and do not realize that there is a separate component which generates the additional elements on top of the animation.

Our preliminary results indicate that adding annotations to program animations encourage novice programmers to view animations more carefully, or at least they spend more time especially in steps which have an-
notations. The eye-tracking experiment described in Publication II gives evidence that the increased time usage is caused by the fact that novices likely read the annotations. Our current research results cannot confirm that using Kelmu would increase learning results and this research should be continued.
Part IV

Discussion

In this last part of the thesis, we discuss in Chapter 12 the contributions and how Learners, Instructors, and Developers will benefit from the outcomes. We also discuss the validity of the research in Chapter 13. In Chapter 14 we discuss how the systems we have presented can be used in other kinds of processes than described in this thesis. We give recommendations in Chapter 15 on how to take advantage of the contribution of this thesis. Finally, in Chapter 16, we give ideas for the future work.
This chapter discusses the contributions of this thesis. We use the familiar roles of Learners, Instructors, and Developers to go through the benefits. We discuss the technical improvements as well as the preliminary results of using the systems in real context.

As already defined in the Introduction, the main motivation of this thesis is to support the overall process of producing online learning content. The three technical contributions represented in the Parts I–III provide technical means for streamlining the overall process, and in this way, providing a better infrastructure for learning.

To recap the overall process, Figure 12.1 visualizes the various phases.

Figure 12.1. The overall process of producing online learning activities as described in this thesis.

We have introduced Jsvee to create generic program animations in Part I, Acos to host online learning activities in Part II, and Kelmu to tailor interactive animations in Part III. These systems cover the whole process of first creating the generic content, making it reusable and available, and finally adding the pedagogical context to the content. The Instructor’s original idea of a code example becomes an interactive program animation for Learners, who use the content as part of the learning materials.

To summarize the main benefits for the three roles, Figure 12.2 uses a similar structure as Figure 1.2 in Section 1.6 to highlight the main outcomes for Learners, Instructors, and Developers.

Next, we will describe in more detail how the contributions of the thesis benefit each role. We discuss how these systems work together to improve the possibilities of creating interactive materials for online learning and
how the systems have improved the field. We start with Developers as many of the benefits are technical. These technical improvements help Developers to create that kind of content which Instructors want to use in their courses.

### 12.1 Developers’ Point of View

For Developers, creating language-specific program animations in expression-level with Jsvee and making these program animations as well as other learning activities reusable with Acos are the main contributions. We focus on these two aspects in the next sections.

#### 12.1.1 Creating Program Animations

To create program animations, we presented the Jsvee library in Chapter 4. Because Jsvee is a library, using it is not as easy as using Online Python Tutor, for example, to produce animations. So why use Jsvee?

Instead of providing a complete visualization application, the aim of Jsvee is to provide those basic functionalities we described in Section 3.5. Jsvee provides those basic components which are required to implement program animations to be used in an online environment. Jsvee is a good choice for those Developers who want to create expression-level program animations to be embedded in online materials because libraries or systems to achieve this did not exist earlier. Jsvee provides a JavaScript-based framework which works in any modern browser without any pre-
requisites such as the Abode Flash plugin or Java Applet, which may also cause security threats.

But especially Jsvee benefits those developers who want to customize or extend the feature set of an animation engine. As Jsvee is a library, it has been designed to be modular and extensible. Implementing new operations to the notional machine of Jsvee or other activities such as custom exercises during an animation can be achieved by developing plugins to Jsvee.

The library nature of Jsvee can benefit all developers who use Jsvee. When the library is improved, all developers using the library can take advantage of the new features. In the same way, extending the existing language packs or creating new packs benefit a broader number of developers. This is more useful than the case that many developers build similar systems but they are totally independent projects. This of course requires an active community to maintain and extend the Jsvee-related components.

12.1.2 Using Acos to Create Reusable Content

We introduced the current problems for distributing online learning activities in Section 6. From the developer’s point of view, the main focus should be in implementing the activities instead of thinking about how the activities can be embedded in learning environments and communicate with LMSs.

Although interoperability protocols, mainly LTI [46], exist, using them is not trivial. Developers must, for example, handle all kinds of tasks related to authentication, validating the requests, storing the user information, as well as the URL to communicate with the LMS, and so on. This is irrelevant when developing content in which the pedagogical reasoning should be the main focus, instead of struggling with the technical details on how to make the content interoperable and reusable.

By using the Acos architecture, the complexity of the interoperability protocols is hidden from developers who create the content. In Acos, the strong separation between the content and communication makes it possible that Acos takes care of the interoperability issues and handles the communication with the LMS.

The server provides a uniform API for communicating with Acos which is always the same regardless of the protocol which is used between Acos and the LMS. For example, sending the grade to the LMS is achieved by
simply calling a function that takes the grade and feedback. Knowing any
details, such as the submission URL, user identity, session id, etc., is not
relevant because the protocol layer of Acos automatically handles all of
these details. This is a solution for the problem presented in Publication
V to provide a proxy that could handle the interoperability issues and help
DEVELOPERS with the technical parts.

Someone might be wondering why to select the Acos approach instead
of using LTI directly, because supporting the Acos API still requires some
work. Although the main benefit is hiding the protocol complexity from
DEVELOPERS, the Acos architecture has more benefits. Because the Acos
API is the same regardless of the interoperability protocol in use, the same
content can be used with multiple interoperability protocols. When the
content is designed to support Acos, the same content can be used with
any interoperability protocol which Acos supports without any modifica-
tions to the content. This increases the reusability of the content as the
LTI protocol is not the only option.

And even if there would be only one interoperability protocol, Acos is not
only a proxy to manage the communication, it is also a server application
to host the content which can be grouped into easily distributed content
packages. Acos architecture also allows running server-side tasks which
may interact with the content. These could include, for example, server-
side grading and server-side compiling.

One important feature that can be achieved with Acos is data collection.
Content can send arbitrary logging events to the Acos server to store any
interaction data that can be used for research purposes later. Therefore,
Acos is not just another way of doing the same tasks which the LTI proto-
col could do, because LTI only provides the support to launch an activity
and submit a grade back to the LMS.

### 12.2 Instructors’ Point of View

In this section, we discuss how INSTRUCTORS can use especially Jsvee and
Kelmu to produce and use interactive content. By using these systems to-
gether, INSTRUCTORS can create pedagogically designed content for online
materials.
12.2.1 Language-specific Animations

For instructors, Jsvee offers a way to visualize CS1 concepts at the expression level. In addition to that, Jsvee supports programming language specific visualizations.

For example, the concept of variables is fundamentally different in Python and Java. In Java, a variable is a memory location which can be allocated but never used. It is possible that there is a variable which is never a target for an assignment. In Python, however, a variable is always a mapping between the variable and its value. Assigning a value to a variable is the only way to create a variable. This may sound like a minor detail, but when teaching the basics, it is important that an animation visualizes the execution correctly.

Language-specific features allow creating animations as described in the previous example. When the Java language pack is in use, an empty variable is created when the variable is introduced and it will remain empty until an assignment occurs. The Python language pack creates the variable only when the value to be assigned to the variable is evaluated.

If we compare this approach with ViLLE presented in Section 2.3.3, the implementation is different. In ViLLE the visualization is always the same regardless of the programming language because it does a direct transformation from one programming language to another. It cannot visualize this kind of differences and is capable of visualizing only the intersection of programming languages which are common in all languages.

By using developers to implement new features to Jsvee’s language packs, animations can truly contain features which are idiomatic in a particular programming language. In this way, instructors can use program animations more widely and be sure that the visual representation is accurate and suitable for the visualized language.

12.2.2 Providing Pedagogical Context by Tailoring

To allow improving general animations and adding a pedagogical context to them, we introduced the Kelmu toolkit in Chapter 10. Kelmu gives new possibilities to emphasize details of the animations by adding textual explanations and arrows on top of the original animation on those steps which contain the relevant parts in that particular context.

In addition, to enhance learning by following Mayer’s multimedia learning principles [71], augmenting an animation integrates it better with the
other materials as the explanations can refer to, for example, a previously introduced concept or other examples. Tailoring can also change the original stepping by combining steps or providing voluntary shortcut paths to skip steps if the Learner is already familiar with them.

Multiple instructors have used the existing Jsvee visualizations, but a few have used Jsvee with the Python transpiler to create new animations and augmented them with Kelmu, too. This shows that the tools provide value for Instructors who can create new content or make the existing content better. The initial feedback for using Kelmu has been promising, which implies the tool is easy enough to use and that there are Instructors who can see the benefits and are willing to try it out. Implementing a new Python visualization with the current toolset for Jsvee and augmenting it with Kelmu takes about 15–30 minutes depending on how many steps there are to be tailored.

12.2.3 Producing and Consuming Online Materials

For Instructors who produce materials, Acos provides one way of creating content repositories which contain online learning activities. The general idea of Acos is to host reusable activities which work in multiple learning management systems. Having repositories containing activities would help Instructors easily enhance their course materials with interactive elements.

By using Acos, Instructors who produce materials can also share their activities with those Instructors who can be seen as consumers, and in this way, increase co-operation with other Instructors within the same subject. Co-operation increases the quality of materials if there are multiple instructors who use them and can provide suggestions how to improve the materials.

12.3 Learners’ Point of View

In the context of introductory programming, using the systems presented in this thesis can provide more a interactive learning experience for novice Learners compared to traditional textbooks or online materials containing only static elements. Both Jsvee and Kelmu aim at better and deeper understanding. As Acos is an invisible technical component for Learners, we omit Acos in this section.
Jsvee provides expression-level program visualizations which can be embedded into course materials and can be used with a modern web browser without any additional plug-ins. Using Jsvee animations is therefore easy for learners.

The expression-level in Jsvee motivates learners to think about the concepts and language constructs in more detail. As the no-function-in-structure principle [26] states, understanding the structure is vital in order to apply small pieces in larger programs to fully understand and construct larger programs. The expression-level visualization gives a visual way to understand that a line of code consists of many small pieces which form together the desired action. The expression-level visualization also aims at giving a robust mental model for learners to think about the dynamic nature of programs. Not many program visualization tools provide expression-level visualization as it is technically challenging to retrieve enough data about the execution at this level.

Our results in Publication III give evidence that learners find program animations useful as they have voluntarily viewed the animations. Also, the feedback we have received highlights that animations used together with text can help with the understanding as the animation and text support each other.

Having Kelmu-tailored animations can make the animations more effective. By following the multimedia learning principles [71], instructors can tailor the materials so that learners better understand which parts of the animation are essential and explanatory texts can improve the understanding of what the animation is visualizing.

Kelmu also has a feature to support multimodal augmentations. In addition to visual annotations or instead of them, a learner could be able to listen to an explanation and follow the visual representation if the instructor has created a narration, for example. This would maximize the processing capabilities of the learner as humans can process verbal and visual elements simultaneously.

The initial results in Publication II give some evidence that tailoring the animations changes student behavior and that students likely pay more attention to the highlighted steps. We assume that this may lead to better understanding and learning results, although the presented data in this thesis cannot validate this assumption.
13. Research Validity

The traditional method of evaluating the credibility of research results is to use two major criteria originally published by Campbell [21] in 1957. These are internal validity: “did in fact the experimental stimulus make some significant difference in this specific instance?” and external validity: “to what populations, settings, and variables can this effect be generalized?”

13.1 Internal Validity

To ensure the quality of the quantitative results in Publication III and Publication IV, we have used automatically generated log traces. In this way, we can get exact information on how students use the program animations during the course, and use this data in the analysis. However, as noted in the Limitations section of Publication III, this data does not provide any details about the learning aspects. Although we noticed in Publication II that Kelmu annotations increase the time-on-task and slow down novices, the data we have cannot answer how the learning results have improved, if they have. Also, the datasets we have cannot describe in what kinds of contexts students have used the animations or what they thought when they viewed the animations.

To triangulate the effects of using Kelmu, we have used three different types of research in Publication II to analyze how Kelmu affects student behavior: log traces, the eye-movement experiment, and the learning effect experiment. Especially the eye-movement data has given valuable additional information on how learners behave when they use Jsvee animations with Kelmu annotations.
External validity is about generalizing the results, will similar results be obtained if the same research setting is repeated elsewhere. Our results in Publication III suggest that students will use program visualization if available, even if no exercise points are awarded. However, in Publication IV results are different. Students used visualizations much less in that context. Also Alvarado et al. [5] have reported that in their context, the usage of program visualization was as that high as in Publication III.

These observations imply that the willingness of using program animation can depend on many factors, such as the way of integrating animations to the course, student background, their motivation and attitude towards programming, other available materials and resources, etc. We cannot say, for example, that using Jsvee program animations would always encourage students to view them. More research should be carried out to determine which factors affect the willingness of using program animation. However, our results show that at least in some contexts in which the program visualizations are tightly integrated with teaching, students will find the animations useful.

The effectiveness of Kelmu toolkit has not been currently researched outside Aalto University. Therefore, we cannot say how the tailoring of animations affect students in other contexts. However, as the Kelmu toolkit follows Mayer’s empirically supported multimedia learning principles [71], it encourages us to expect that using Kelmu can enhance learning if the tailoring is properly done.

We do not have statistically significant results of learning gains of using Jsvee and Kelmu together. Therefore, we can only say that in our experiments, the student behavior was different when the Kelmu annotations were present but we cannot say how much they affected learning and would that be generalizable.

Although the tools presented in this thesis have been used in real university level courses, we consider the research as pilot studies to get initial results on the usefulness and effects of using the tools. In order to give more generalizable results, more research with a larger number of students from multiple countries and contexts should be part of the upcoming research settings.
13.3 Technical Considerations

Evaluating the technical architecture of Jsvee, Acos, and Kelmu is not an easy task. One of the major limitations is that there are few external developers who are familiar with the systems. New content for Acos and integrating Kelmu with other animators than Jsvee have been under construction, and the feedback from developers has been mainly positive. We are not aware of external developers using Jsvee. Therefore, as the number of developers is low and the architecture is not very well evaluated by other developers, we cannot say much about the quality of the technical decisions.

However, all the three systems have been in real use to provide contents for multiple CS1 courses which have in total more than 2000 students. This indicates that the systems are mature and can be used without any major problems in production use. Animations for Jsvee exist in Java, Python, Scala, and C#. The language pack for Python is the most comprehensive but creating language packs for other languages has proven that the architecture with the language-independent core with the ability to extend it and create language-specific external language packs works well. Implementing new content types for Acos has also been easy for external developers which indicates the working architectural solutions.
14. Supporting Other Kinds of Processes

In this thesis, we have described the process of using Jsvee, Acos, and Kelmu together in this order. However, this is not the only possibility. All the three systems are separate components and can be used individually or in other processes.

For example, if there is no need for integrating content with an LMS, it is possible to use only Jsvee and Kelmu together and place the content to a separate webpage for example. Of course, if only a generic animation is enough for some use cases, Jsvee can be used alone without Kelmu.

It is also possible that the instructor who designs the program animations also tailors them before publishing them. Then there would be available at least one augmented version of the animation. In some cases, having some predefined annotations can be better than no annotations at all, because in this way the instructor who designed the animation can describe the original pedagogical context for the animation. A consuming instructor can check whether the annotations are suitable, or extend or replace them with another annotations which are better for another pedagogical context.

Acos does not have any dependencies with Jsvee. If there are some other interactive materials which would be distributed and shared via Acos, that is possible as long as the materials are written in JavaScript so that the materials can interact with the server. Creating new content types for Acos provides a way for creating reusable materials which can be used with multiple LMSs. The package architecture of Acos allows an easy distribution method for the content type packages. If there are, for example, multiple choice quizzes or other content that could grade the submission, Acos is an efficient solution to implement the required communication between the content and the LMS.

We have shown Kelmu to be used only with Jsvee but other animators
are supported as well, if they support the requirements explained in Publication II. This means that using Kelmu with, for example, JSAV algorithm animations or Online Python Tutor animations is possible without any changes to these systems. Using Kelmu with new animators can enhance learning in other contexts than programming as well because many disciplines use different kinds of animations that would be suitable for adding more pedagogical context to them.

The loose technical integration between the three components enables these variations as well as using Acos and Kelmu also with other interactive content than Jsvee. This gives the flexibility to one, two, or three systems from this thesis and design how the benefits they provide can help work with other interactive materials.
15. Recommendations

Based on our results, we give three recommendations for CS1 teachers who might benefit from the outcomes of this thesis.

**Using interactive program animations can enrich course materials**

Jsvee provides one way of creating expression-level program visualizations. We have been able to get good results by tightly combining animations with other course materials and using animations throughout the course. As learners view animations voluntarily, they must give additional value for learners. Understanding the dynamic execution process which the expression-level visualization aims at gives robust mental models of the execution model and helps to understand the structure and meaning of each piece of the code.

**Tailor animations to improve their pedagogical influence**

Automatically generated animations can have many steps but only a few of them form the essential part of the learning objectives in a particular learning context. By tailoring an animation with the Kelmu toolkit, learners can follow the animation easier and notice the important sections the animation visualizes. Instructors can start the tailoring process from scratch, or improve or change the annotations by the instructor who created the animation. Our initial results indicate that students likely read the annotations and pay more attention to the content.

**Make online learning activities reusable**

If you can produce high-quality online learning activities, make it usable for other instructors, too. Acos architecture allows this kind of interoperability with quite a low effort from developers. Also, reusable activities give the possibility to use content made by other instructors instead of implementing all the content by yourself. Using Acos has been an easy way for developers to create new interactive content and make it interoperable with multiple learning management systems.
Recommendations
16. Future Work

This section discusses how to carry on the research and how to develop the systems in the future.

16.1 Technical Development

Easier Ways to Create Jsvee Animations

One of the most important targets in the future would be to create better tools for creating Jsvee animations more easily. Currently, there is only a transpiler for Python programming language that can produce new visualizations from the given source code. The transpiler supports only a subset of the Python language.

There should also be a similar tool for Java, for example. The lack of these transpilers hinders the usage of Jsvee because producing new animations takes too much time without a tool which can create the mapping between the source code and the operation sequence for Jsvee’s notional machine.

Implementing a new transpiler requires a parser which can parse the source code and produce the corresponding abstract syntax tree (AST). By traversing the AST, the transpiler can emit operations for the operation sequence. If there is a parser available for the desired language, implementing a simple version of a new transpiler can be probably implemented within a week. Extending the transpiler takes more time because each AST node must have a separate handler to convert the node to corresponding operations. The current version of the Python transpiler was implemented within a few weeks.
More Language Packs for Jsvee

At the moment, Jsvee language packs exist for Python, Java, C#, and Scala. They have limited support for the emulated programming languages. There should be more language packs, such as for C and C++, which are also popular languages in CS1 courses. Also, the existing language packs should be extended to cover more built-in API features.

Implementing a completely new language pack is not very hard, but requires defining many essential features of the language. For example, how the arithmetic operators work in that language and what are the data types and values they produce. It must also be defined how these operations are emulated with JavaScript if this is the way how the language pack implements operators.

Making built-in functions available typically means that the original function is emulated with JavaScript. Therefore, adding a new function may be possible by using similar features in JavaScript or otherwise a new implementation must be written. Depending on how similar functionality exists in JavaScript, emulating a function may require less than ten lines of code, which makes it rather easy and fast. Instead of implementing a vast subset of the language, the current language packs are constructed for the local needs and could be extended to cover more functionality.

Using a Real Execution Environment in Jsvee

In the future, a prototype using a real interpreter on the client side or on the server side should be tested. For example, there is a JavaScript version of CPython which runs in a browser. Using a real interpreter would make implementing new language packs easier and cover more built-in features. There are some technical problems to solve first, such as how to synchronize the view and the program state with the interpreter if the executed instruction has side effects as the state of the visualization should be the same as the internal state of the interpreter.

Improving Kelmu Editor

The present Kelmu Editor as shown in Figure 10.2 can be used to augment animations. However, the experience of using the editor is not as good as it could be. For example, functions such as copy and paste are missing and editing HTML formatted explanations as raw text can be difficult. Also,
the visual appearance of the editor is more like a prototype than a ready product.

Furthermore, there could be more annotation types available, such as different kinds of arrows and shapes. It should be possible to anchor these annotations to the HTML elements instead of using exact coordinates to allow at least some changes in the appearance of the animation.

The current implementation of Kelmu requires that the animation always proceeds exactly in the same way. The editor and library should support branching so that the next annotations are selected based on some criteria, or the underlying animator could send a message about the current execution path and annotations would be selected accordingly.

### 16.2 Future Research

**Experiments About Learning Effects of Jsvee and Kelmu**

In the future, there should be more controlled experiments on how using Jsvee and Kelmu affects learning results.

For example, we know that students use program animations in the observed CS1 course in which the visualizations are an integral part of course materials, but we still do not know how much they can learn from the animations. The experiment where two animations were augmented and the students were asked to predict the output of a similar program could not give statistically significant results as the number of participants was too low. More research should be carried out to discover in which situations augmented program animations can enhance learning.

**Learners as Instructors**

Currently, Learners using Jsvee have been the end-users who view the visualizations designed by Instructors. One possible future research area would be to have Learners as Instructors. It would be interesting to see what kind of animations learners would implement to teach some concepts for other Learners, or what kinds of annotations Learners would create. And would these activities enhance learning as well?

As Learners often know what was difficult for them to understand, can they create somehow different examples with Jsvee than Instructors? Or if a Learner only worked with Kelmu, would the augmentations look
Future Work

different? In this kind of setting, measuring the learning effect can give interesting results because teaching something to others requires deeper understanding of the topic.

Creating Acos and Jsvee Communities

We have created the Acos architecture to enable easier creation of online learning activities. We would like to see more content developers and instructors using Acos. The idea of an online repository of exercises needs more content types and a broader selection of activities.

Having an active community around the architecture would enable developing Acos further. Using Acos at multiple institutions would also give new opportunities to collect more data about using the learning activities. The more data, the more interesting research in the field of learning analytics can be carried out.

Of course, there is a challenge that there must be enough active INSTRUCTORS and DEVELOPERS in the community in order to make it work and keep the available resources available and up-to-date. Such communities also exist for some tools. There is, for example, the Greenroom community [31] which produces and delivers content for the Greenfoot environment.

All the same apply to Jsvee as well. It would be great to see that Jsvee animations would be part of even more CS1 courses. If there were more courses using Jsvee, it could provide more developing resources to extend the language packs, create new language packs and work with the transpilers. Acos would provide an optimal solution to distribute the animations and give the ability to use Kelmu to augment the produced generic animations.
Appendix A contains a full list of the ready-made operations to manipulate the notional machine of Jsvee. Operations usually take one or multiple parameters to define the action. Typically, one operation is executed per step but it is possible to execute operations before or after the actual operation during a single step. Operations to precede the actual operation are prefixed with an underscore (for example _disableAnimations), and operations to be executed after the actual operation are postfixed with an underscore. Description ending with an asterisk (*) denotes an action which changes the internal state but does not produce a visible operation.

**addCollectionInitializer** Adds a new collection initializer to the evaluation area.

**addFunction** Adds a new function call to the evaluation area.

**addFunctionFromVariable** Adds a function pointer from the specified variable to the evaluation area.

**addFunctionReference** Adds a function pointer pointing to the specified function to the evaluation area.

**addOperator** Adds a new operator to the evaluation area.

**addReference** Adds a reference to the evaluation area.

**addValue** Adds a literal value to the evaluation area.

**addValueFromField** Adds a value from a specific instance variable to the evaluation area.

**addValueFromVariable** Adds a value from a specific local variable to the evaluation area.

**alert** Shows an alert pop-up with pre-defined text.

**alertOnce** Shows an alert pop-up with pre-defined text but only once if this operation is inside a loop, for example.
Built-in Operations in Jsvee

**assign** Assigns the value in the evaluation area to the defined variable.

**assignField** Assigns the value in the evaluation area to the defined instance variable.

**assignFields** Copies multiple values in local variables to the defined instance variables.

**assignParameter** Assigns a parameter value to the defined variable.

**assignParameters** Assigns parameter values to the defined variables.

**beginBlock** Begins block-scoped section of variable definitions*.

**clearEvaluationArea** Clears the evaluation area.

**conditionalJump** Branches the execution based on the value in the evaluation area (typically the next operation changes the current line as this instruction only changes the internal program counter)*.

**convertFunctionToCallable** Converts a function pointer in the evaluation area to a callable function.

**createArray** Creates a new array to the heap.

**createClass** Creates a new class to memory.

**createField** Creates a new field to the specified instance.

**createFields** Creates multiple fields to the specified instance.

**createFrame** Creates a new call frame to the stack.

**createFunction** Creates a new function to memory.

**createInstance** Creates a new instance to the heap.

**createIterator** Creates a new iterator to iterate a collection.

**createOperator** Creates a new operator to be used during the execution.

**createParameterVariables** Creates parameter variables.

**createValue** Creates a new literal value.

**createVariable** Creates a new local variable.

**deleteState** Deletes the previous undo state*.

**disableAnimations** Disables the transition animation between steps*.

**disableStepping** Executes operations until enableStepping*.

**enableAnimations** Enables the transition animations*.

**enableStepping** Executes only one operation per step*.

**end** Indicates the end of the animation.

**endBlock** Ends the current block-scoped variable section and deletes all variables created inside this section.
**evaluateFunction** Begins the evaluation of the specified function call in the evaluation area.

**evaluateOperator** Evaluates the specified operator in the evaluation area.

**flashElement** Flashes a particular element to draw user's attention.

**getValueAtIndex** Fetches a value from the specified collection in the given index.

**hideInfo** Hides the shown additional information.

**initializeCollection** Initializes a collection with initial values inside collection initializer in the evaluation area.

**iterate** Advances the specified iterator*.

**jumpFalse** Sets the currently active line in the code panel (used with conditional jumping to get a descriptive explanation).

**jumpIterationReady** Sets the currently active line in the code panel (used with iteration to get a descriptive explanation).

**jumpTrue** Sets the currently active line in the code panel (used with conditional jumping to get a descriptive explanation).

**label** Defines a named position in the operation sequence (not an executable operation)*.

**nop** No-operation instruction*.

**removeVariable** Removes the specified variable from the stack frame.

**returnValue** Returns the value in the evaluation area as a result from the active call.

**returnVoid** Returns an empty value as a result from the active call.

**runForward** Executes the specified number of operations.

**setLine** Sets the currently active line in the code panel.

**setValueAtIndex** Sets a value to the specified index in the specified collection.

**showInfo** Shows information as static text below the animation.

**showText** Shows the specified text as current status text.

**takeNext** Takes the next value from the collection to be iterated.


References


References


Creating, Tailoring, and Distributing Program Animations

Supporting the Production Process of Interactive Learning Content

Teemu Sirkiä