A review of program visualizations and design of a visual dataflow language

Marin Aglić Čuvić
Dept. of Informatics
Faculty of Science
Split, Croatia
maragl@pmfst.hr

***Abstract*—Different types of visualizations have been developed to assist novices with learning to program. These include program visualization systems and visual programming languages, among others. Program visualizations are used to represent how a program is executed by the notional machine; an abstract machine formed from the concepts of a programming language. Visual programming languages (VPLs), on the other hand, attempt to simplify learning to program by providing visual representations of programming concepts. They visually represent the structure of a program, and sometimes visualize its execution. We focus mostly on dataflow visual programming languages as a subcategory of VPLs. We provide a literature review on program visualizations from 2013 to July 2019. Afterwards, we take a look at different taxonomies of VPLs and describe dataflow visual programming languages. The paper also discusses a visual dataflow language for learning programming and a program visualization system for actor programs.**

Keywords—actor programs, education, dataflow visual language, program visualization, review

I. INTRODUCTION

Numerous studies have reported that learning to program is difficult. Programming requires novices to possess complex problem-solving skills as well as learn the syntax and logic of programming constructs. Introductory programming courses often have a set of learning outcomes that students need to achieve. A typical CS1 course may teach students concepts such as variables, branching, loops and functions. Novices are then expected to master each concept to some degree which they demonstrate by solving problems.

It is suggested that course difficulty depends on "how much a student should be able to achieve" for its duration (Luxton-Reilly, 2016). Lower teacher expectations or teaching fewer concepts may make introductory programming easier. However, lowering expectations could lead to a situation where students' do not master the concepts at a level required by a future course. Reducing the number of concepts in a CS1 course may mean moving them to a later course which may introduce problems of its own. Additionally, students that are not enrolled in computer science, but have some computing course, may already be taught only the basics.

Much research in programming education is concerned with improving the ways to teach introductory programming. Such studies may be divided into five categories: theories, orientation, delivery, tools and infrastructure (Luxton-Reilly et al., 2018). Additionally, the choice of programming language for a first course is the topic of much debate (Stefik & Hanenberg, 2014). Visual programming languages (VPL) have been developed to reduce the effort of learning programming syntax and errors related to it. In this paper, we are interested in visual programming languages (VPLs) and generic program visualization (PV) systems. Their aim is to reduce the barrier of learning to program.

Literature reviews provide a way to inform researchers and educators of the current trends and tools available to improve their teaching and students' learning experience. Therefore, we conducted a literature review of program visualization systems. We discuss each tool with regard to the engagement taxonomy provided by Sorva, its features and evaluations (Sorva, Karavirta, & Malmi, 2013). We do not include program visualization systems that have been recognized as inactive in previous literature reviews.

Existing engagement taxonomies for PV systems are discussed in this paper. Each taxonomy defines certain engagement levels that describe different ways learners can engage with the visualization. Higher engagement levels should often require greater cognitive effort and lead to better learning (Naps et al., 2002).

Most PV systems share a certain set of features, such as stepping through the execution and highlighting the current instruction. We developed a dataflow programming language (DFVPL) for introductory programming that supports some of the features found in PVs. The system was later extended to support the demonstration of programs written in the functional programming paradigm. We used the system in an introductory and functional programming setting and provide a brief report on our experiences.

Finally, we describe a system for visualizing the interaction between actors in the actor model. The system can also be extended to visualize the communication between other types of entities such as objects and agents. The results of the pilot study we conducted seem promising (Čolak & Čuvić, 2019).

The remainder of this paper is organized as follows. Section II gives theoretical background about the notional machine, and section III discusses visualizations and their relationship to the notional machine. In section IV we give a brief overview of existing engagement taxonomies. Section V discusses previous literature reviews on program visualizations, and section VI describes our methodology. Program visualization systems, their features and evaluation are described in section VII. Visual programming languages are discussed in more detail in section VIII with a focus on DFVPLs. Section IX introduces the visualization systems that we have developed and describes our experiences and studies with them. Section X describes recent DFVPLs with application in data science. Section XI discusses our conclusion and describes future work.

II. THE NOTIONAL MACHINE

Rather than just writing computer programs, programming includes reading, understanding, and tracing the execution of already written code. These skills enable programmers to find bugs, improve the existing codebase, and add new functionality. However, programmers require a mental model of the system to do so. The same is true for novices.

Mental models are mental structures that one possesses about a system (Sorva, 2013). In the case of programming, that system is the machine that executes the program. Mental models are runnable, which allows programmers to use them to simulate program execution in memory.

Useful mental models are ones that are at an appropriate level of abstraction (Hidalgo-Céspedes, Marín-Raventós, & Lara-Villagrán, 2016; Sorva, 2013). If a model is at a low level of abstraction, it may not be trackable due to many variables and states. A model at a high level of abstraction may exclude essential information. Programmers reason about a program at the level of abstraction provided by the programming language. These abstractions form a new machine, that is called a *notional machine*. Different programming languages can have different notional machines. Furthermore, a single programming language is not limited to a single machine. E.g., an object-oriented programming (OOP) language may have two notional machines, one for reasoning about the behavior of a method inside an object, and another to reason about the interactions between objects. Some programming languages, such as Scala, which support OOP and functional programming (FP) may have an additional notional machine. This machine might allow us to reason about a program in terms of data transformations in a pipeline of functions.

Novices need to acquire a correct mental model of the notional machine to write correct programs (Sorva, 2013). However, their models are often faulty, based on superficial language features, and contain misconceptions. We define a misconception as an incomplete or incorrect understanding of a programming concept. Du Bouley (1986) found that the causes of most misconceptions are faulty understandings of the notional machine. While the formation of mental models is intuitive, correcting existing ones is significantly more difficult (Schumacher & Czerwinski, 1992). The problem is that people feel comfortable with the mental model they already have, despite its potential flaws. Solving a programming problem successfully with a faulty mental model may reinforce a novice's belief in its correctness. Therefore, it is necessary to provide a correct model of the machine as soon as possible.

Visualizations of the notional machine may assist novices with the construction of valid mental models (Sorva, 2013). They provide a concrete view of how the notional machine executes a program. A teacher may choose to draw visualizations on the blackboard or use a program visualization tool. Many different tools exist which support visualizing the execution of one or more programming languages.

Another way to deal with misconceptions is to introduce cognitive conflict. Cognitive conflict refers to challenging novices' existing non-viable mental models in order to encourage them to recognize the errors in them and seek improvement (Ma, Ferguson, Roper, Ross, & Wood, 2009). Ma et al. (2008) proposed four stages of using cognitive conflict for teaching programming with program visualizations. Those stages include: 1) identify typical inappropriate models, 2) challenge existing mental models and push novices into cognitive conflict status, 3) assist novices with the construction of viable models, and 4) allow novices to solve programming problems on their own.

Visual programming languages may also be useful to alleviate some of the difficulties when learning about the notional machine. Their graphical representations have a much simpler syntax and provide a clearer view of a program's structure. Additionally, some visual languages have a stage which is used to visualize the execution of a subset of instructions, like Scratch for example (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010).

III. VISUALIZATIONS

Visualization refers to the use of graphical elements to represent information and other phenomena (Čuvić, Maras, & Mladenović, 2017; Hidalgo-Céspedes et al., 2016). They are often used to facilitate the formation of mental representations of complex and abstract phenomena (Sorva, Karavirta, et al., 2013). Some applications include data, knowledge, and educational visualization. A visualization that is used to represent some aspects of software is called a software visualization (SV).

Several taxonomies for SVs have been proposed (E. Lahtinen & Ahoniemi, 2005; Maletic, Marcus, & Collard, 2002; Myers, 1990; Naps et al., 2002; Price, Baecker, & Small, 1993; Roman & Cox, 1993; Sorva, Karavirta, et al., 2013; Stasko & Patterson, 1992). Some of them focus on SVs in general, while others focus on those used to teach programming. For example, the taxonomy introduced by Maletic, Marcus, & Collard (2002) uses five dimensions to describe SVs based on their support for software development and maintenance. Price, Baecker, & Small (1993) a taxonomy that consists of categories and subcategories organized into a multi-level n-array tree. Their taxonomy is extensive and designed to be extendable. Lahtinen & Ahoniemi (2005) introduced a taxonomy for introductory programming course visualizations based on their support for different levels of Bloom's taxonomy. Naps et al. (2002) introduced a taxonomy for educational visualizations based on their supported level of engagement.

In this paper, we adopt the classification introduced by Price et al. (1993) and adopted by Sorva (2013). The field of software visualization contains two broad subfields: algorithm visualization (AV) and program visualization (PV). Algorithm visualization tools represent algorithms at a high level of abstraction. They are often independent of the programming language and targeted towards advanced programming courses. Program visualization tools visualize concrete programs on a lower level of abstraction. They often target novice programmers. The adopted classification recognizes two subfields of PVs: visualizations of static code and runtime visualizations. We focus on PVs that visualize the execution of programs by the notional machine.

Included into this categorization is visual programming. Visual languages enable the specification of programs using visual techniques (Price et al., 1993; Sorva, Karavirta, et al., 2013). They can be used to specify code and represent runtime dynamics. Therefore, they are included as a subfield of both static code visualization and runtime dynamics visualization. For example, it is possible to specify code in Blockly and

visualize which instruction is executed ("Blockly," n.d.). We will refer to executable visual languages as *visual programming languages* (VPL). When adopting the classification introduced by Price et al. (1993), Sorva (2013) also added *visual programming simulations* (VPS). VPS is a subtype of runtime dynamics visualizations. It allows the user to manipulate graphics in order to simulate the execution of a program by the machine (Sorva & Sirkia, 2010).

For completeness we mention the SV classification used by Hidalgo-Céspedes et al. (2016) in their literature review on PVs. They classify SVs into three main types based on the representation's level of detail: code visualization, algorithm visualization and program visualization. Therefore, code visualization is considered one of the main fields, rather than a subfield of PVs. Additionally, they report that the target audience for code visualizations are professional developers, while PVs target novice programmers.

In this paper, we are interested in those PVs that visualize the execution of a program by the notional machine. Going forward, when talking about program visualizations we refer to program visualization tools that visualize the execution of the notional machine.

IV. ENGAGEMENT TAXONOMIES

The components of a visualization and its level of abstraction determine what is possible to learn from it (Sorva, Karavirta, et al., 2013). They also influence its applicability for learning different content. However, passively viewing a visualization may not produce the expected impact on learning. Instead, learners must actively engage with the visualization to benefit from it (Ben-Ari, 1998).

Hundhausen et al. (2002) conducted a meta-study of AV effectiveness. They noticed that 71% of studies that included active engagement of learners reported significant results. In contrast, only 33% of studies that did not actively engage learners reported significant results. The authors concluded that the way learners engage with the visualization has a more significant impact on learning than its visual representation. These findings motivated the introduction of engagement taxonomies for software visualizations.

Naps et al. (2002) introduced the first SV engagement taxonomy with a focus on AVs. For consistency with Sorva's literature review, we refer to this taxonomy as the original engagement taxonomy (OET). The OET introduced six levels of engagement: no viewing, viewing, responding, changing, constructing, and presenting. The levels represent increasingly engaging forms of interaction. They do not form strict hierarchies and overlap between levels above viewing is possible (Naps et al., 2002). The introduction of OET motivated further research focused on comparing the effectiveness of different engagement levels (see, e.g., Urquiza-Fuentes & Velázquez-Iturbide [2013], Banerjee et al. [2013; 2015]). The general consensus is that a higher level of engagement will have a more significant impact on learning.

Algorithm and program visualizations do not always support the same kind of user interactions. Because the OET was based on AVs, it does not include some forms of engagement typical of PVs. Myller et al. (2009) noticed this and introduced the extended engagement taxonomy (EET). The EET includes additional levels: controlled viewing, entering input, modifying and reviewing.

Sorva et al. (2013) found certain shortcomings with the levels of OET and EET. They addressed these shortcomings by introducing a two-dimensional taxonomy (2DET). The first dimension is the direct engagement dimension. They argue that OET's constructing level contains two types of activities that do not pose an equally challenging cognitive task. These two types of activities are separated in the 2DET into creating and applying. In this paper, we refer to those two activities as constructing *a* and *b* when talking about the OET. Furthermore, the presenting and reviewing levels of EET refer to equally challenging tasks. These are combined in the 2DET to presenting.

The other dimension of 2DET is the content ownership dimension, which considers the relationship between the learner and content that is visualized. Two level of the EET also consider this relationship – entering input and modifying. Sorva et al. (2013) believe that this relationship influences the capability of learners to map the content to its visualization and their motivation to engage with the visualization. Table 1 contains all of the levels of the three engagement taxonomies.

Despite the newer taxonomies that are proposed, it seems that OET is still predominantly used when comparing the effectiveness of different engagement levels. There may be a few reasons for this. One may be that most studies do not compare engagement levels above responding, or even viewing. Since the taxonomies greatly coincide on these lower levels, there is no need to use a finer grained taxonomy. Another reason may be that the OET has more abstract levels, which leads to more activities being included in a single level. This may be a consequence of drawing inspiration from AV research. EET's categorization system is somewhat dubious in its details (see Sorva 2013). Educational research usually takes certain time, which may mean that researchers simply did not have a chance to use the 2DET.

Whatever the reason, we categorize the visualization tools and the reviewed evaluations with regard to the 2DET. Because most research on comparing the effectiveness of different engagement levels is based on the OET, we provide a table which sums up the relationship between the direct engagement dimension of the 2DET to other engagement taxonomies (Table 2). The table also provides explanations of why we believe these relationships hold.

TABLE 1 ENGAGEMENT TAXONOMIES

#	OET levels	EET levels	2DET	
			Direct engagement levels	***Content ownership levels***
1	No viewing	No viewing	No viewing	Given content
2	Viewing	Viewing	Viewing	Own cases
3	Responding	Controlled viewing	Controlled viewing	Modified content
4	Changing	Entering input	Responding	Own content
5	Constructing	Responding	Applying	
6	Presenting	Changing	Presenting	
7		Modifying	Constructing	
8		Constructing		
9		Reviewing		
10		Presenting		

TABLE 2 RELATIONSIHP BETWEEN LEVELS OF DIFFERENT ENGAGEMENT TAXONOMIES

	2DET	OET	EET	Description	Reason
Direct engagement dimension	No viewing	No viewing	No viewing	No visualization is used	Intuitive
	Viewing	Viewing	Viewing	The learner views the visualization with little or no interaction	Intuitive
	Controlled viewing	Viewing	Controlled viewing	Besides just viewing the visualization, the learner can step through the execution, control the visualization speed and inspect its components.	OET's viewing level corresponds better to controlled viewing since less cognitive effort is required than for responding.
	Responding	Responding, changing*	Responding, entering input*, modifying*	The learner responds to questions about the visualized content. Can occur during or after the visualization finishes.	Responding levels are intuitive. But the entering input level can be considered equivalent if a student is asked a question to enter input that will result with some behavior of the program - similar reasoning as for changing in OET, described by Naps.
	Applying	Constructing (a)	Changing	The learner modifies the visualization to perform a task. An example is manipulation of visualization components.	The first constructing activity described by Naps et al. (2002) can take the form of manipulating the visualization to simulate the execution of an algorithm. EET specifies changing as manipulating visualization elements.
	Presenting	Presenting	Presenting, reviewing	The learner presents a detailed analysis or description of the visualization, potentially to an audience.	Sorva (2013) argues that the EET uses two categories of equally challenging tasks and these may be combined in a single category.
	Constructing	Constructing (b), presenting*	Constructing	The learner creates his own visualizations of the target software.	The OET's constructing level describes an activity that requires users to construct their own visualizations. For the presenting level, it is equivalent if the learner creates his own visualization.

* conditional equivalence

We note that it is difficult to relate the content ownership dimension to some of the levels in the OET and EET. However, the EET does include *entering input* and *modifying* levels. These two levels would correspond to the *own cases* and *modified content* levels in the content ownership dimension respectively. Hence, it would make no sense to relate them to the direct engagement dimension, which is why they are not included in Table 2.

V. PREVIOUS PROGRAM VISUALIZATION REVIEWS

A comprehensive literature review on program visualizations was conducted by Sorva et al. (2013). Their review focuses on generic program visualizations of runtime dynamics whose aim is to assist in learning and teaching. The authors provide a brief summary of each tool as well as the studies and results obtained from those studies of each tool. Their review also includes information about the notional machine elements each visualization supports, evaluation method, and supported programming paradigm. The review covers a period from the 1980s to 2013 and identifies 46 different visualization tools.

Hidalgo-Céspedes et al. (2016) wrote a review to update the program visualization list with tools that emerged from 2013 to 2016. Their review includes only active visualization tools. However, they did not provide a summary of the studies carried out with the tools or a comprehensive discussion on their evaluation and supported notional machine elements. They also did not discuss the engagement levels with regard to the 2DET. Rather, the authors evaluated each tool based on a set of constructivist principles.

Therefore, we base our review on that done by Sorva et al. (2013) since it is more comprehensive. We include the tools reported by Hidalgo-Céspedes et al. (2016).

VI. SCOPE OF PROGRAM VISUALIZATION REVIEW

We use a systematic literature review approach to provide an updated list of program visualization tools with regard to the elements of the notional machine they support, evaluation method and the 2DET. The review method is based on those used in reviews carried out by Al-Sakkaf, Omar, & Ahmad (2019) and Berney & Bétrancourt (2016). It inclues a description of the search query and databeses searched, eligibility criteria and overview of the selection process.

Our review includes generic program visualization tools that represent the execution of a program by the notional machine, with the intent of providing a learning aid for novices and teaching aid for teachers in introductory programming. With regard to the classification provided by Maletic et al. (2002) (discussed in section III), this defines the *task*, *target*, and *audince* of the PVs. We do not pose any restrictions on the *representation*, but the *medium* has to be an electronic device and the visualization available onscreen (Sorva, Karavirta, et al., 2013).

The search string contains varying terms that refer to PVs. For instance, sometimes the term "animation" is used instead of "visualization". Additionally, the term SV was sometimes used in the past to refer to PVs. We also limited the search to include only those visualizations that are intended for education. Our final search string was:

```
("software animation" OR "program animation" OR
"software visualisation" OR "program visualisation" OR
"software visualization" OR "software visualization" OR
"visual debugger") AND (educ* OR teach* OR learn*)
```

The search covered papers that were published between 2013 and July 2019. We searched the following databases: (1) Web of Knowledge, (2) ACM Digital Library, (3) Scopus, and (4) IEEE Xplore. Results of each database were downloaded

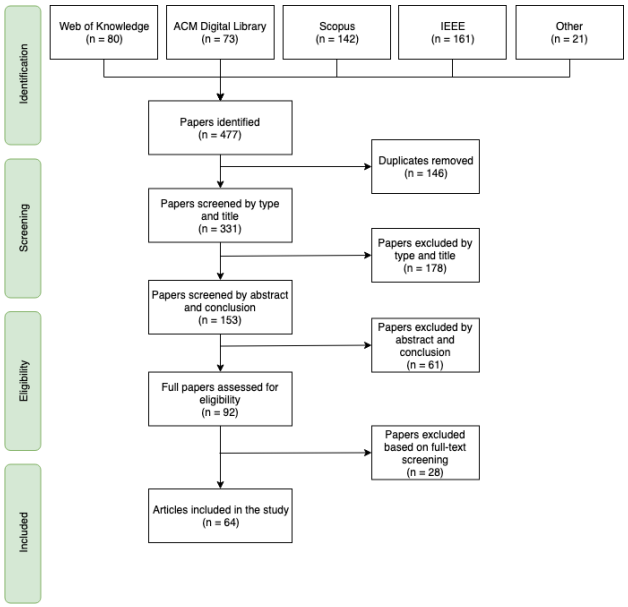

Fig. 1 Paper selection process

and imported into an excel spreadsheet. All papers were subject to the inclusion/exclusion criteria presented in Table 3. The selection process was done in three stages. First, some of the papers were excluded based on their title and type. For instance, we excluded posters, papers records that had only abstract, and panels. Secondly, the abstract of the remaining papers was read and assessed for eligilibility. In some cases, the conclusion was also read and assessed. Finally, the papers were assessed based on screening their full text. We excluded papers that presented inactive PV systems that were alrady reported in other literature reviews, e.g. Evizor (Moons & De Backer, 2013). The remaining papers are included.

Additionally, our review includes papers that were identified through scanning the references and citations of eligible papers, and other sources. However, we kept the criteria that the paper is published on or after 2013. We refer to the source of these papers as Other. Fig. 1 depicts the search process.

The goal of this review is to provide an updated list of PV tools. A PV tool is included in our list if it is active, available or has not been included in previous literature reviews. We consider a PV tool active if there is evidence that the tool is still being actively developed, maintained or used to publish new research. Other PV tools are not discussed in this review.

Several active PV systems have been used for research well before the publication range set up in this paper. To provide a complete picture of how the systems were evaluated, we include these papers in our discussion. Most of these have also been included in previous literature reviews.

In the next section, we will describe new program visualization tools that have appeared in our search. We will also discuss their evaluations and results.

VII. Program visualizatin systems

In this section, we will provide a description of each PV system and its capabilities. To better structure our review, we distinguish two types of research papers: 1) system development, and 2) evaluation. System development papers are those that refer to the PV system itself, the addition of new features to it, or development of other systems that extend its capabilities in purposeful new ways beyond visualization. Evaluation refers to those papers that have evaluated the visualization aspect provided by a PV system. Therefore, we will first discuss each PV system and papers related to development. Afterwards, we review the research related to their evaluation. Some papers may belong to both types of papers.

TABLE 3 INCLUSION AND EXCLUSION CRITERIA

Insertion criteria	Exclusion criteria
Paper presents a new generic PV tool as discussed in this review	Paper presents a specialized or already reported inactive PV tool
Paper presents an extension of an existing PV tool	Paper presents a PV tool for a non general purpose language
Paper evaluates the use of a generic PV tool	Paper presents a new SV or PV with a focus on non-introductory programming or industry
Paper introduces a new taxonomy for PVs	Paper refers to algorithm visualization
Paper presents a visual debugger that is similar to PV tools discussed in this review	Paper refers to PV tools for static code analysis
Paper is a literature review on PV	Paper refers to other educational tools that are not PVs as discussed in this paper
	Paper presents a tool that uses an existing PV system
	Paper is not a full paper, i.e. poster, panel, abstract only
	Paper refers to other tools or techniques for making visualizations, e.g. libraries
	Paper uses existing PV tool to propose new methodology
	Full proceedings as they occur as a record in some database searches
	Paper is not in english

A summary of the tools is presented in tables X and Y. The tables include information relevant for teachers and researchers.

It is worth noting that a literature review on student engagement with SVs was recently published. The review extracts different theoretical foundations for developing SVs and proposes several design principles for future SVs. The review covers the period from 2011 to 2017. However, it does not provide a review of the results of comparing the effectivneses of different engagement levels.

The rest of this section is divided into the following subsections:

A. Legend for tables.
B. Controlled viewing of given content
C. Controlled viewing of own cases
D. Animating own content with no control
E. Controlled viewing of own or modified content
F. Own content with controlled viewing and responding
G. Applying or creating visualizations of own content
H. Models for program visualization
I. Evaluation of systems

A. Legend for tables

In this section we will describe the legend for Table 4 and Table 5. Table 4 contains general information about the PV systems. Table 5 contains more specific information about the visualization components, engagement level and how the system was evaluated after 2013. The tables are based on those

provided in the previous two literature reviews (Hidalgo-Céspedes et al., 2016; Sorva, Karavirta, et al., 2013).

1) Legend for table 4

System name is the name of the system. In cases where the system is not named, the authors name is provided. *Supported programming language* and *programming paradigm* contain the information about the programming language and paradigm that the systems support. Programming paradigms are *imp* for imperative, *OO* for Object-oriented and *func* for functional.

The *installation* column contains information on how to install the visualization. Web based visualizations do not require installation. Systems can have a standalone installation or as a plugin for NetBeans or Eclipse. *Platform* mentions on which platforms the PV system can be installed. If a system is web based, then the value for platform is *web*. Most systems can be installed on all of the major platforms: Linux (L), Mac (M) and Windows (W).

Most of the PV systems can be obtained via the web. If a PV system can be obtained this way, the column *available* contains the link to the download site. Otherwise, the system is unavailable or can be obtained in some other way, which is mentioned in the table. *Status* refers to our best guess whether the system is active or inactive. *At least since* contains the year in which the system seems to have first appeared, either in literature or when the first version was released (Sorva, Karavirta, et al., 2013).

2) Legend for table 5

The *Notional machine elements* column contains a list of some of the key notional machine elements that the system can visually represent. *Vars* indicates the visualization of variables, *Refs* the visualization of references and/or pointers, *Addrs* memory addresses, and *Objs* stands for objects. *Classes* means that the system can visualize classes as part of the runtime, and not just as part of static class diagrams. *Struct* refers to any composite data such as arrays, lists, records, etc. *Control* refers to the visualization of the active part of the program at each stage of execution. *ExprEv* indicates that the system can visualize the process of expression evaluation (Sorva, Karavirta, et al., 2013).

Content ownership and *direct engagement* dimensions indicate the position of a system on the 2DET. For the content ownership dimension, only the highest level is listed since systems supporting own content implies that the system can also support modified content and others. Given content is listed only if the system provides a distinct mode for example given content. For direct engagement, the viewing level is not explicitly mentioned unless it is the only level supported (Sorva, Karavirta, et al., 2013).

Step grain indicates the granularity of what gets executed and visualized at each step. Only the lowest level is listed in the table. *Statement* (S) indicates that at each step, an entire statement or declaration is executed. *Expression* (E) indicates that the system supports visualizing expression evaluation in detail. *Message passing* (MP) refers to object interactions and communication between different entities.

Evaluation (≥ 2013) indicates the ways in which the visualization provided by a PV system was evaluated. Some studies did not specify that students were novice programmers, and some explicitly mention that participants were students enrolled in non-introductory programming courses. However, we included them if they evaluated the visualization aspect of a PV system and are relevant for the discussion of included PV systems. We include only evaluations published in articles on or after 2013. Experimental, qualitative and survey evaluations are included. We describe each of these types of evaluations in section I (Evaluations). We did not find any anecdotal evaluations during our literature search.

B. Controlled viewing of given content

Systems in this category allow users to control the execution of the visualization of given content. These systems require teachers to prepare the examples in advance and learners have no control over the visualized content. Only one new PV system is placed in this category: EDPVE.

EDPVE stands for Example-based dynamic program visualization. The system supports the visualization of predefined program examples written in Pascal. It shows the state of all variables in the program, as well as a flowchart view of the code. Learners can execute the program line-by-line, with the current line highlighted both in code and flowchart windows. The system also provides information on the line currently being executed. Learners can interact the visualization only by providing input and viewing output of the example program. The system is a research prototype developed for the purpose of conducting a research study (Tekdal, 2013). To the best of our knowledge, the PV is not active.

C. Controlled viewing with own cases

The systems discussed in this section allow the user to control the execution of the visualization and provide their own input. We have not recognized new systems that fall into this class. Hence, all systems reviewed were still considered active in previous literature reviews.

PlanAni is a PV system that supports visualizing the execution of short programs (Sorva, Karavirta, et al., 2013). The system visualizes variables and operations with them based on the variables' roles. Roles are a cognitive concept derived from the way variables are used. In his study of roles, Sajaniemi (2002) found that 99% of variables found in novice-level procedural programs can be classified into nine roles. For each role, PlanAni uses a specific metaphor for visualization. For example, a stone is used to visualize a constant, while a temporary variable is visualized with a "flashlight that is on as long as the value is used", a stepper is visualized with footsteps (Jorma Sajaniemi & Kuittinen, 2003).

The visualization metaphors introduced by PlanAni were later extended to include OO concepts (Jorma Sajaniemi, Byckling, & Gerdt, 2007). These include metaphors for classes, objects, object references, method invocation, parameter passing, return value and garbage collection. The metaphor-based OO animator is a collection of Flash animations of predefined examples (Jorma Sajaniemi, Byckling, & Gerdt, 2006). The learner is able to provide his own cases (Sorva, Karavirta, et al., 2013).

Both PlanAni and the metaphor-based OO animations seem to be inactive. Their web sites have not been updated in over eight years. Although a paper which evaluated the use of PlanAni was published relatively recently, when considering other factors, we are unsure if the system should be considered the system active.

TABLE 4 PV SYSTEMS GENERAL INFORMATION

#	System name (or author)	Supported languages	Prog. paradigm	Installation	Platform	Available	Status	At least since
1	Grasph/ jGrasp	Java, C, C++, Objective-C, Ada, VHDL	imp, OO	standalone, eclipse plugin	M, L, W	https://www.jgrasp.org/	active	1996
2	Jeliot 2000/Jeliot 3	Java	imp, OO	standalone	M, L, W	http://cs.joensuu.fi/jeliot/	active	2003
3	JIVE	Java	imp, OO	eclipse plugin	M, L, W	https://cse.buffalo.edu/jive/	active	2002
4	Metaphor-based OO Visualizer	Java	OO	flash or swf player required	Web, M, L, W	http://saja.kapsi.fi/oo_metaphors/	inactive	2007
5	Online Python Tutor (OPT)	Python, C/C++, JavaScript, Ruby, Typescript, Java	imp, OO	not required	Web	http://pythontutor.com/	active	2010
6	PlanAni	Pascal, Java, C, Python	imp	standalone	M, L, W	http://www.cs.uef.fi/~saja/var_roles/planani/index.html	inactive?	2002
7	The Teaching Machine	C++, Java	imp, OO	not required	Web	http://www.theteachingmachine.org/	inactive	2000
8	UUhistle	Python	imp, OO	standalone	M, L, W	http://www.uuhistle.org/index.php	inactive	2009
9	Ville	various	imp	not required,	Web	https://ville.utu.fi/	active	2005
10	VIP	C++	imp	standalone	M, L, W	http://www.cs.tut.fi/~vip/en/	active	2005
11	WinHIPE	Hope	func	standalone	M, L, W	http://www.lite.etsii.urjc.es/tools/winhipe/	active	1998
12	BlueJ Novis	Java	imp, OO	BlueJ	M, L, W	https://www.bluej.org/	active?	2013
13	FM /TM Visualization	Language independent	imp	not software	Not software	Not software	active	2014
14	JavelinaCode/ JaguarCode	Java	imp, OO	not required	Web	http://www.jaguarcode.org/	active	2015
15	Jeliot ConAn	Java	imp, OO	standalone	M, L, W	http://cs.joensuu.fi/jeliot/	active	2013
16	PROVIT	C	imp	not required	web	http://cleast.u-aizu.ac.jp/introduction-0/index-introduction.html	active	2014
17	SeeC	C	imp	standalone	M, L, W	https://seec-team.github.io/seec/	inactive?	2013
18	Virtual-C	C	imp	standalone	M, L, W	https://sites.google.com/site/virtualcide/	active	2014
19	EDPVE	Pascal	imp	standalone	unknown	unavailable	inactive	2012
20	JAD	Java	imp, OO	integrated into Jload	web	unavailable	active	2017
21	LISN	Java, language-independent	imp	standalone	unknown	unavailable	inactive?	2018
22	ObjectVisualizer	Java	OO	standalone	unknown	unavailable	active?	2017
23	Omnicode	Python	imp	not required	Web	unavailable	inactive	2017
24	PandionJ	Java	imp, OO	pre-installed with Eclipse, Eclipse plugin	M, L, W	http://pandionj.iscte-iul.pt/installation.html	active	2017
25	PITON/DS-PITON	Python	imp, OO	standalone	M, L, W	ask via email	active	2018
26	PVC	C	imp	not required	Web	https://ryoskate.jp/PlayVisualizerC.js/	active	2017
27	SunLab	Java, pseudo language	imp	standalone	Android	unavailable	active?	2018
28	TEDViT	C, C++	imp	standalone	unknown	unavailable	active	2015
29	Thonny IDE	Python	imp, OO	standalone	M, L, W	https://thonny.org/	active	2015
30	(Nagae, Koji)	C	imp	not required	Web	on request	inactive	2013

□ included in 2013 review □ included in 2016 review □ new systems

TABLE 5 VISUALIZATION AND EVALUATION SPECIFIC DETAILS OF PV SYSTEMS

#	System name (or author)	Notional machine elements	Content ownership dimension	Direct engagement dimension	Evaluation (≥ 2013)	Step grain
1	Grasph/jGrasp	Control, Vars, Calls, Refs, Objs, Structs	own content	ctrl'd viewing	S	S
2	Jeliot 2000/Jeliot 3	Control, Vars, ExprEv, Calls, Refs, Objs, Classes, Structs	own content	ctrl'd viewing [responding]	-	E
3	JIVE	Control, Vars, Refs, Objs, Classes, Calls, Structs	own content	ctrl'd viewing	-	S
4	Metaphor-based OO Visualizer	Control, Vars, ExprEv, Refs, Objs, Classes, Calls, Structs	own cases	ctrl'd viewing	-	E
5	Online Python Tutor (OPT)	Control, Vars, Objs, Classes, Refs, Calls, Structs	own content	ctrl'd viewing	E, S	S
6	PlanAni	Control, Vars, ExprEv, Structs	own cases	ctrl'd viewing	E, S	E
7	The Teaching Machine	Control, Vars, ExprEv, Calls, Addrs, Refs, Objs, Structs	own content	ctrl'd viewing	-	E
8	UUhistle	Control, Vars, ExprEv, Calls, Refs, Objs, Classes, Structs	own content	ctrl'd viewing, responding, applying	Q	E
9	Ville	Control, Vars, Calls, Structs	own content	ctrl'd viewing, responding, applying	S	S
10	VIP	Control, Vars, ExprEv, Calls, Refs, Structs	own content	ctrl'd viewing	Q	S
11	WinHIPE	Control, Vars, ExprEv, Refs, Calls, Structs	given content, own content	ctrl'd viewing, applying	E, S	E
12	BlueJ Novis	Control, Vars, Calls, Refs, Objs, Classes, Structs	own content	ctrl'd viewing	-	S
13	FM/TM Visualization	Conceptual spheres [Control, Vars, Calls]	given content	viewing	E	[E]
14	JavelinaCode/JaguarCode	Control, Vars, Objs, Classes, Refs, Calls, Structs	own content	ctrl'd viewing	E, S	S
15	Jeliot ConAn	Control, Vars, ExprEv, Calls, Refs, Objs, Classes, Structs	own content	responding	E, S	E
16	PROVIT	Control, Vars, Calls, Structs	own content	ctrl'd viewing	S	S
17	SeeC	Control, Vars, ExprEv, Calls, Refs, Structs	own content	ctrl'd viewing	-	S
18	Virtual-C	Control, Vars, Refs, Addrs, Structs	own content	ctrl'd viewing, responding	S	S
19	EDPVE	Control, Vars	given content	ctrl'd viewing	E	S
20	JAD	Control, Vars, Refs, Objs, Structs	own content	ctrl'd viewing	E, S	S
21	LISN	Control, Vars	own content	ctrl'd viewing	E	S
22	ObjectVisualizer	Control, Calls, Refs, Objs	own content	viewing	E	MP
23	Omnicode	Control, Vars, [Objs, Classes, Refs, Calls, Structs]	own content	ctrl'd viewing	Q	S
24	PandionJ	Control, Vars, Objs, Classes, Refs, Calls, Structs	own content	ctrl'd viewing	E	S
25	PITON/DS-PITON	Control, Vars, Refs, Calls, Structs	own content	ctrl'd viewing	E, S	S
26	PVC	Control, Vars, Refs, Calls, Addrs, Structs	own content	ctrl'd viewing	E, S	S
27	SunLab	Control, Vars, Calls	own content	ctrl'd viewing	S	E
28	TEDViT	Control, Vars, Refs, Calls, Addrs, Structs	modified content, [own content]	ctrl'd viewing	E, S	S
29	Thonny IDE	Control, Vars, ExprEv, Calls, Addrs Refs, Objs, Classes, Structs	own content	ctrl'd viewing	S	E
30	(Nagae, Koji)	Control, Vars, Calls	own content	ctrl'd viewing	-	S

□ included in 2013 review □ included in 2016 review □ new systems

D. Animating own content with no control

The systems in this category allow learners to animate their own programs. However, no interaction with the visualization or its execution is supported. We found one new system that fits in this category: ObjectVisualizer.

ObjectVisualizer is a system that visualizes the execution of a program by transforming its source code (Shin, 2018). The system visualizes the interaction and relationships between objects in a Java program. It seems to support the visualization of learners' own content based on the implementation details provided. However, it does not seem to provide any engagement above viewing. The visualization is provided simultaneously as the program is executing. The system may still be in active development.

E. Controlled viewing of own or modified content

Systems in this category allow the user to visualize their own programs and control their execution with stepping and/or play commands. Actually, most PV systems support these levels of the 2DET. Because there are many such systems, we group them based on their support for different programming languages: 1) PV systems for C/C++, 2) PV systems for Java, 3) PV systems for Python, and 4) multi-language PV systems.

1) PV systems for C/C++

PROVIT is a PV system for visualizing C programs (Yu Yan, Hiroto, Kohei, Shota, & He, 2014). It can visualize variable values and function calls. Variables are represented with a box that includes its data type, name and value. A variable's visualization is highlighted in blue if the next statement refers to that variable, and in red if the next statement will change its value. During execution, lines of code that are already executed are underlined in blue, and the next statement is underlined with red. PROVIT was initially developed as a desktop application, with a web version created later (Y. Yan, Nakano, Hara, Kazuma, & He, 2016). The visualizations provided by the web version can be embedded into a PowerPoint presentation.

PROVIT-CI is an extension of PROVIT for instructors (YAN, HARA, KAZUMA, HISADA, & HE, 2018). It provides some functionalities convenient for the classroom, such as generating a PROVIT URL for a given example. The user just needs to upload the example source code and input data. PROVIT-CI also extends PROVIT by providing a new array viewer and visualization of return values.

PlayVisualizerC (PVC) is a web-based PV system for the C programming language. It executes programs at the instruction level and highlights the current line in program execution and provides visualizations for variables and stack frames. PVC supports detailed visualization of dynamically allocated memory, file and standard I/O (Ishizue, Sakamoto, Washizaki, & Fukazawa, 2018).

SeeC is a PV system for the C programming language built upon the Clang project (“Clang C Language Family Frontend for LLVM,” n.d.; Egan & McDonald, 2014). It visualizes program execution as a graph, which contains a node for each active function call. Two types of arrows are used, a blue dashed arrow to connect the calling function with the called function, and a black arrow for pointers. A node contains the name of the function and variable values.

The system can automatically generate explanations that are linked to relevant pieces of code (Egan & McDonald, 2014). When a user's mouse cursor hovers over part of the explanation, the system highlights the corresponding piece of code. The explanations may also reference external materials via URL.

SeeC's dynamic evaluation tree visualizes the execution of a statement at the expression level (Egan & McDonald, 2015). The statement is visualized at the top of the tree, with evaluation steps lower in the tree, and the final result at the bottom. The values that are produced by an evaluation step are placed directly below the expressions that produced them. The system also checks for errors typically made by students before an instruction is executed (Egan & McDonald, 2013).

SeeC also allows a rich set of ways to move through the visualization. Users can choose to move to a point before or after a value in memory changes, and to the beginning or end of a function call, to name a few (“SeeC,” n.d.).

TEDViT was initially developed for teaching algorithms, such as sorting (Yamashita et al., 2015). However, the visual representations it provides place it in scope of this review. TEDViT's visualization interface can be divided into two parts. In one part, the memory state of all variables is shown. The authors refer to the second one as "target domain world". The system allows teachers to configure how certain components visualized in the target domain world should be represented. For example, teachers can define whether arrays should be visualized horizontally or vertically. The configuration can be specified via set of rules (T-Rule sets), which in turn can be defined through a separate GUI based system (Tezuka et al., 2016). In the initial version, these rules significantly limited learner's engagement with the visualized content. However, they were later changed to allow for more flexibility (Yamashita et al., 2018). The authors claim that learner's own content can be visualized. However, in their study they gave students source code with predefined variable names to match the configured rules. It is unclear how the target domain world behaves if the names do not match.

The system also visualizes the flow of data between functions (Yamamoto et al., 2017). For recursive functions, learners can choose to observe it as a black box. The system then skips visualizing the behavior of the recursive functions. Learners can also choose to view the behavior of the recursive function for an arbitrary number of recursive calls. Once they wish to stop, a function call that satisfies the termination condition is shown.

The authors also extended TEDViT so that it can visualize the states of a program at two different execution steps (Ihara et al., 2017), which might help learners understand how algorithms work by comparing the state of the program at two different points in time. The system can also visualize the execution of two different programs side-by-side. The rationale for this decision was to allow learners to compare 1) two programs with different data in order to understand program behavior independent of data, or 2) two different programs, e.g. programs with correct and incorrect behavior.

The Teaching Machine is a web-based PV system that can visualize the execution of C++ and Java programs (M. P. Bruce-Lockhart & Norvell, 2007). The system's expression engine supports visualizing expression evaluation in detail (M. P. Bruce-Lockhart & Norvell, 2000). It also allows the learner to step through an expression in a single step and an

unlimited undo facility to step back through some part of the execution. The state of heap and stack memory is also represented. A special linked view visualizes the relationship between data in the stack and heap. The view is suitable for visualizing data structures in an advanced programming course (M. P. Bruce-Lockhart & Norvell, 2000). A later version of the teaching machine also has the ability to generate quizzes on algorithms at a high level of abstraction (M. Bruce-Lockhart, Crescenzi, & Norvell, 2009; Sorva, Karavirta, et al., 2013). This capability places the system on the responding level of the engagement taxonomy. However, because the quizzes are on algorithms, the highest engagement level supported for introductory programming is controlled viewing. The system was not used in recent publications and does not seem to have gotten an update in quite some time.

VIP is a PV system that supports the visualization of a subset of the C++ language (Isohanni & Knobelsdorf, 2013). The system is based on the Clip interpreter ("CLIP," n.d.). It visualizes variable values and highlights them as they change. Whenever the system executes an expression, it shows the values of operands, operators and the resulting subexpressions. Teachers can also prepare examples that will be shown during execution.

Nagae & Kagawa (2014) introduced a prototype visual debugger for the C programming language. However, the system's interface is in Japanese. It can visualize call stacks, bitwise operations and variables. It allows teachers to specify alternative representations of visualization components. The system seems to be developed purely as a research prototype and is no longer maintained.

2) PV systems for Java

BlueJ Novis is an extension of the BlueJ IDE for providing a visualization of the notional machine (M. Berry & Kölling, 2016; Michael Berry & Kölling, 2013, 2014). It supports the visualization of Java programs at different levels of detail. At the highest level of detail, the system visualizes all object fields, method call chains, and parameter and return value passing. Lower detail levels show objects in a simplified view with references to other objects and method invocations. The lowest detail level shows a heatmap which displays object activity. A step in Novis is a method call or return. However, it supports statement level step granularity via BlueJ debugger. Novis does not seem to be included in the downloadable version of BlueJ, therefore we are unsure if it's still active. We did not find any papers that evaluated BlueJ Novis.

JAD is a visual debugger for Java designed to assist deaf and hearing-impaired (DHI) students (Nascimento et al., 2017). Despite its focus on DHI students, it might be used in introductory programming courses. JAD is embedded in JLoad, a Java e-learning object for the deaf, which is embedded in a learning platform (Silva, Oliveira, Oliveira, & Freitas, 2014). JAD provides controlled execution and can visualize variables. The system also provides sign language explanations for error messages (Nascimento et al., 2017).

JaguarCode, formerly known as JavelinaCode is a web-based PV system for Java (Jeong Yang, Young Lee, & Hicks, 2016; J. Yang, Lee, & Chang, 2017). JaguarCode provides both static and dynamic visualization for OO programs (Earwood, Jeong Yang, & Young Lee, 2016). Static visualization provides three sets of UML class diagrams (J. Yang, Lee, Hicks, & Chang, 2015). One for the active Java program, one compact that shows the relationship between classes in the project, and a detailed diagram with all project information. The system supports adding new files to an active project. JaguarCode also seems to be able to generate object and sequence UML diagrams (J. Yang, Lee, Gandhi, & Valli, 2017).

When a line of code is executed, its corresponding class in the UML class diagrams is highlighted. JaguarCode can visualize variable values, function calls, objects and their instance variables. Learners can step forward and backwards through program execution. Stepping through is at the statement level. The system is still under development and is currently available only to authorized users. Future plans for JaguarCode development are discussed by Yang et al. (2017).

Jeliot 3 is the culmination of years of research that started with Eliot (Sorva, Karavirta, et al., 2013). It is one of the most well-known and researched family of PV systems. Its predecessors include Eliot, Jeliot I and Jeliot 2000 (Ben-Ari et al., 2011; Haajanen et al., 1997; S.- Lahtinen, Sutinen, Tarhio, & Tuovinen, 1997; Levy, Ben-Ari, & Uronen, 2003). Each system attempted to improve on the one before. The first system, Eliot, visualized variables of C programs. Jeliot was a proof of concept web-based visualization system that visualized Java programs. However, it had a GUI that was too complex for beginner programmers. Jeliot 2000 introduced a simpler interface and more complete animations.

Jeliot 3 added the ability to visualize classes and objects (Moreno, Myller, Sutinen, & Ben-Ari, 2004; Sorva, Karavirta, et al., 2013). The visualization area is divided into four areas for visualization components of different type. Jeliot 3 supports visualizing a large subset of the Java language. Surprisingly, as opposed to many other PV systems, Jeliot 3 does not support stepping back through the program. The system visualizes control, variables, expression evaluation, method calls, references, objects, classes and various structs. mJeliot is a tool that allows learners to answer questions about the visualization on their mobile phones (Pears & Rogalli, 2011). Learners also receive feedback about their response. mJeliot may alleviate Jeliot 3 to the responding level of the direct engagement dimension.

jGrasp is a lightweight development environment that provides various visualizations for several programming languages ("JGRASP Home Page," n.d.), including C++ and Java. It can visualize variables, arrays, and objects and their states. The system provides viewers that can display an object at different levels of abstraction (Hendrix, Cross, & Barowski, 2004). Higher-level viewers are more appropriate for learning about data structures, such as linked lists and trees. The object viewer provides a lower-level view of an object's state.

Version 2.0 of jGrasp introduced the new viewer canvases (Cross, Hendrix, Barowski, & Umphress, 2014). Learners can *play* the viewer canvas in order to auto-step, i.e. animate, the execution. The learner can choose additional details for a viewer, such as array indices, and host multiple viewers in a single canvas. For example, in case of learning how a sorting algorithm works, the learner can add an array viewer and a bar graph viewer in a single canvas, which allows multiple visualizations of the algorithm.

JIVE is an interactive environment suitable for novice-level programs and larger object-oriented and multithreaded programs (Sorva, Karavirta, et al., 2013). It is developed as an Eclipse plugin and provides multiple views of execution,

including object and sequence diagrams (P. Gestwicki & Jayaraman, 2005; P. V. Gestwicki, 2004). Both of these diagrams can be shown in a detailed and compact view. The system also provides a *call-path view* which provides a compact visualization of a series of method activations.

Recent research with JIVE has dealt with compacting the visualization of sequence diagrams (Jayaraman, Jayaraman, & Lessa, 2017) and the introduction of state diagrams (Ziarek, Jayaraman, Lessa, & Swaminathan, 2016). Jayaraman, Jayaraman, & Lessa have (2017) introduced horizontal, vertical and hybrid techniques for compacting sequence diagrams and compaction techniques of the execution of multi-threaded programs with different forms of interaction. State diagrams were introduced in order to provide a more concise way to visualize execution and state information (Ziarek et al., 2016). The system also can also check the consistency between a runtime and design-time provided state diagram. The consistency check does not test for equality. JIVE highlights states and transitions in the runtime state diagram that are not included in the design-time diagram, which may point to possible bugs in implementation.

PandionJ is a visual pedagogical debugger for Java (Santos & Sousa, 2017). The system offers control commands typical for debuggers, such as *step in*, *step out*, *step over*, *resume*, and *stop* execution. It can visualize class, objects, variables, structs, and control (Santos, 2018). PandionJ also allows learners to interact with visualized objects and invoke their public methods. One of the goals of the system is to somewhat mimic the way teachers might draw visualizations of variables. Therefore, the system analyzes the user's code in order to assign certain roles to some variables.

PandionJ offers a widget extension that allows alternative visualization for primitive values, arrays and objects (Santos, 2018). To use alternative visualizations, called *widgets*, of primitive values and arrays, the user needs to annotate their declaration with the appropriate tag. The system can automatically associate an object type with a widget.

SunLab is a PV system for Android smartphones. The initial version supported the visualization of Java programs but has since changed to a pseudo language Dynamic program visualization on android smartphones for novice Java programmers (E. Kumalija, Yi, & Fatih, 2018). It supports the visualization of variables, function calls and expression evaluation. The system can also interact with smartphone sensors (E. J. Kumalija, Fatih, & Sun, 2019).

3) PV systems for Python

CodeSkulptor is a web-based programming environment for Python. I was developed a massive open online course (MOOC). The system incorporates OPT's visualization capabilities. However, OPT does not seem to support event-driven GUI programs, whereas CodeSkulptor does. Certain types of events may be automatically triggered numerous times which may be difficult to visualize. To remedy this, when CodeSkulptor detects events, such as draw and ticks, it adds buttons that will allow the user to manually trigger them (Tang, Rixner, & Warren, 2014),

Omnicode is a prototype IDE built upon Online Python Tutor (OPT). Omnicode is live, which means that any changes to the code are automatically visualized. The system uses scatter plots to visualize the entire history values of numeric variables. The visualization is limited in this regard. However, the system does take advantage of being built upon OPT. After a user selects a line of code, a detailed visualization is provided. Users can also create their own scatterplots to display visualizations of other meaningful numeric values, e.g. length of a list (Kang & Guo, 2017). Omnicode was a graduate project that is not being actively developed or used.

PITON is an IDE for Python that combines PV and programming workspace (Elvina, Karnalim, Ayub, & Wijanto, 2018). It combines features of Online Python Tutor (Philip J. Guo, 2013) and PyCharm ("PyCharm," n.d.). Major features taken from OPT are visualization of step-by-step execution, variable visualization, and control highlighting. Minor features that enhance the usability of the system are also implemented, such as highlighting variables whose value changes and simplified errored messages, PITON also provides different mechanisms for providing program input. The classic mechanism which accepts input when the corresponding input instruction is executed. The second and third mechanisms allow the user to provide all inputs before the code is executed, one of which accepts inputs from a file. Major PyCharm features added to PITON include compile & run, source code highlighting and file manipulation. Programs in PITON may be run without invoking the visualization.

DS-PITON is an extension of PITON which includes an AV tool (Nathasya, Karnalim, & Ayub, 2019). The extended system can visualize seven different data structures in its data structure display. The display can show multiple data structures simultaneously. The system can also show the code implementing the data structures and visualize the execution of their methods.

Thonny is another IDE for that supports the visualization of Python programs. The system supports stepping into and over program statements. When the learner steps over a statement, Thonny executes the whole statement. In case the learner decides to step into a statement, Thonny highlights the first child of the statement, if there is one. Stepping into also enables users to evaluate expressions step-by-step. Function execution is visualized in its own window with its own stack frame. Heap values are represented with an ID, which when selected shows the values associated with it. The system also logs user activities in a file, which enables the reproduction of user actions and program construction (Annamaa, 2015).

4) PV systems that support multiple languages

Online Python Tutor (OPT) is another long-lived and popular PV system (Philip J. Guo, 2013). It supports the visualization of programs several programming languages, including Python and Java. OPT is a web-based system whose visualizations can be embedded into other web pages. It supports visualizations at the statement level. Over the years, additional systems have been built on top of or based on OPT. Systems that were built on top of OPT include Codepourri, Codechella, and OPT+Graph. Others that were based on OPT include TraceDiff and a system introduced by Azadmanesh and Hauswirth.

Codepourri is a tool that enables users to annotate source code to create tutorials (Gordon & Guo, 2015). These tutorials are embeddable into other web pages. Annotations can be created at each step of the execution. Users can also see annotations left by others and vote on the best one.

Codechella is a system embedded with OPT that allows collaborative learning (P. J. Guo, White, & Zanelatto, 2015). Users can start shared sessions and join via a generated URL. All members of a session see the same programming code and

visualization. These are kept synchronized, regardless of which member edits the code or advances the visualization. Members can also see each other's mouse pointers. The system also provides a joined chat for all session members.

OPT+Graph is a tool based on OPT for visualizing graph data structures in C (Dien & Asnar, 2018). The tool supports bar graph, list and tree representations.

Azadmanesh and Hauswirth (2017) introduced a system based on OPT that provides more information about statement execution and intuitive code explanations. They changed the backend of OPT and modified its frontend. The system supports visualization of Java programs. It visualizes expression evaluation in a tree-like structure. Their visualization retains the information about the order in which expressions are executed and shows the intermediate and final values of evaluation. Values presented lower in the structure are fetched and evaluated first, and the final result is on top. Each step is accompanied by a spoken explanation provided by a JavaScript speech Synthesizer.

TraceDiff is a tool based on OPT that highlights the difference in the execution of a programs actual and expected behavior (Suzuki et al., 2017). A learner submits a program to the TraceDiff system, which identifies potential corrections and synthesizes a correct program. The system extracts the differences between the two and highlights them on the user interface.

LISN is a language-independent PV system. It was primarily developed as a prototype to introduce and demonstrate an embedding technique for language-independent PV tools. To use the system, the educator needs to provide the target source code and two feature sets. The first feature set is related to the programming language, and the second to the source code itself. The programming language feature set includes the run and compile commands, input, output, state, and executable file names. The source code feature set contains the source code, and library import, file writer invocation and file writer declaration instructions (Sulistiani & Karnalim, 2018).

Hence, in case a programming language or source code change, one or both new feature sets need to be provided (Sulistiani & Karnalim, 2018). The system operates in three phases. In the first phase, the source code is embedded with library import instructions, file-writer declaration instructions and file writer-invocation instructions. The source code, embedded source code, run, and compile commands are then passed to the second phase which compiles and runs the code. It also generates a file containing the visualization states. The states file is then passed to the visualization phase that visualizes it. The system can visualize control and variable states. It is currently not being further developed.

F. Own content with controlled viewing and responding

The systems in this section provide the user the freedom to step through the execution, and possibly answer questions about the visualization.

Jeliot ConAn is a version of Jeliot 3 that produces conflictive animations (Moreno, Sutinen, & Joy, 2014). These animations include incorrect visualizations of certain steps (Hidalgo-Céspedes et al., 2016; Moreno, Sutinen, Bednarik, & Myller, 2007). Their purpose is to keep the learner in a state of cognitive conflict for the duration of the visualization. The learner needs to actively think about what he sees and respond when he believes an incorrect visualization occurred.

Moreno et al. (2013) proposed a game concept based on conflictive animations. In their proposed concept, students should use a PV system to create conflictive animations and challenge their peers to find the conflict. The students creating the conflictive animations will have to identify the concepts he had most problems with. Students should use a separate system for the resolution of conflicts, which would also be used a repository for the animations. The peers reviewing the animation need to find the step of the conflict and identify the concept that was incorrectly animated. Students would be awarded points if they correctly solved the conflicts. There is a possibility to have correct animations in which case students should indicate that the animation is correct.

Virtual-C IDE is an educational development environment for the C programming language (D. Pawelczak & Baumann, 2014). Variables are visualized in memory blocks with colors corresponding to the type of memory. Plugins for Virtual-C can be implemented with JavaScript. An example of a plugin that visualizes data structures is given by the authors (see Pawelczak & Baumann [2014]).

The system provides a testing framework (TF) that enables teachers to create tests that evaluate users' knowledge (D. Pawelczak & Baumann, 2014). It also provides static and dynamic tests for user code (Dieter Pawelczak, Baumann, & Schmudde, 2015). An example of a static test is to check whether a function is correctly invoked. Dynamic tests include I/O, performance and function tests, which are predefined. The teacher can also write assertion and expectation tests. A test suite file needs to be provided for the generation of test cases.

G. Applying or creating visualizations of own content

In this section, we cover systems that support engagement similar to visual program simulation. The exception is WinHIPE, which allows learners to configure animations themselves.

UUhistle is a well-known PV system for visualizing Python programs (Sorva & Sirkia, 2010). The system supports visualizing own content and provides an explanation of what is happening with the visualization for each step. The learner can step through the execution in expression or step level. It also provides a *visual program simulation* (VPS) mode of execution. In VPS, the learner manipulates the visualization to execute the program with a mouse. The learner is required to execute each instruction in the appropriate order and use and allocate program memory. The system also allows learners to make mistakes and gives feedback on them.

UUhistle is currently not being actively supported or developed as noted on its website. We are not aware if it is used in teaching. Therefore, we assume it is inactive. Additionally, two successor systems for UUhistle, Jsvee and Kelmu, have been introduced (Sirkiä, 2016). Jsvee is a language angostic library for program visualizations. It can assist educators with creating visualizations of the notional machine. Kelmu is a toolkit for augmenting generated animations with additional elements, such as textual explanations. However, these systems are out of scope of this review. We refer the interested reader to Sirkiä & Sorva (2015), Sirkiä (2016) and Hosseini et al. (2016) for additional information.

Ville started out as a PV system with support for multiple programming languages, including Java and Python (Laakso, Kaila, & Rajala, 2018). The new version of Ville is a collaborative learning platform. We are only interested in its visualization aspect for this paper.

In earlier versions, the teacher needed to provide examples which could then be modified by the user (Kaila, Rajala, Laakso, & Salakoski, 2008; Sorva, Karavirta, et al., 2013). However, a recent paper suggests that learners can write full programs. The teacher only needs to create the appropriate exercise (Laakso et al., 2018).

Ville supports multiple types of automatically assessed exercises for programming and provides an interface for implementing new ones in Java. Users may be required to write programs or modify existing ones. Teachers can add popup questions to example programs, which will appear at certain steps in the visualization (Kaila et al., 2008). Students may be given a shuffled program, which they are required to re-arrange to produce a working solution. Another type of exercise engages students in simulating the execution of a program. The system provides certain components which students use to manually execute certain aspects of a program (Laakso et al., 2018; Sorva, Karavirta, et al., 2013).

The visualization can be viewed in a step-like fashion or animated with adjustable speed. Ville supports the visualization of function calls and variables. It offers a parallel mode which visualizes two different programming languages simultaneously. Users can add support for new programming languages via built-in syntax editor (Kaila et al., 2008).

WinHipe is an IDE that supports the visualization of the functional programming language HOPE. The IDE is based on term rewriting, an evaluation model of functional programming. In one control step, the users can evaluate the entire expression, perform *n* rewrite terms or evaluate a reducible expression. Users can also construct animations using WinHIPE. The user types in and evaluates an expression by using the actions he believes are most appropriate, which generates a set of visualizations. He then selects the visualizations that will be used in the animation. The animations can be played in WinHIPE or used to generate a web page along with the source code and descriptions (Pareja-Flores, Urquiza-Fuentes, & Velázquez-Iturbide, 2007). The system's facility to construct program animations can also be used by students. Students can be given a problem and the solution source code, and required to provide suitable input data, choose the most appropriate visualization steps and write a description of the solution. Therefore, WinHIPE may be placed on the applying level of the direct engagement dimension (Sorva, Karavirta, et al., 2013; Urquiza-Fuentes & Velázquez-Iturbide, 2012).

H. Models for program visualization

Here we include models that were introduced for PV. These have not been implemented as a software system. There is currently only a single model that falls to this category.

TM Visualization, previously called *FM Visualization*, is a visualization model for describing the structure of a system (Al-Fedaghi & Alrashed, 2014; AlFedaghi, 2019). It has not yet been implemented into any PV system. The model describes a system in terms of components and flows. Components are organized into conceptual spheres, which may intersect with or encompass other spheres. The model was introduced to include various software and hardware aspects of program execution that are most often ignored in PVs. Depending on the program, the visualization may include different conceptual spheres, e.g. arithmetic logic unit (ALU) if the program executes an arithmetic operation. Statements themselves are represented with conceptual spheres (Al-Fedaghi & Alrashed, 2014). The information in tables X and Y is based on the examples of C++ programs provided by Al-Fedaghi & Alrashed (2014). As opposed to other visualizations, TM visualization seems to show the entire execution process as a graph, at least for some programs.

The model can be used to describe other types of systems, not just programs. Some examples include a half-adder, vending machine, and a water phase diagram (Al-Fedaghi & Sultan, 2017).

If a hypothetical system were to implement TM visualization, we believe that its step grain would be similar to that of expression evaluation. Since expression evaluation contains subexpression that may also have subexpressions, this would correspond to spheres which contain other spheres. Higher levels of step grain may be supported as *n* flow/sphere steps, which would be similar to WinHIPE's *n* rewrite terms (Pareja-Flores et al., 2007).

I. Evaluations

Many of the PV systems discussed in the previous sections were evaluated to collect student feedback on the system or evaluate their effectiveness on student learning. In this section, we review the publications identified through our literature search that evaluate the effect of visualization on learning to program. We do not consider evaluations of other aspects of a PV system.

We include experimental, qualitative, anecdotal and survey evaluations. Experimental evaluations may include quasi-experiments or experiments with quantitative data. We have a special interest in evaluations regarding different levels of engagement taxonomies. Qualitative evaluations are those that refer to rigorous qualitative research methods. Anecdotal evaluations are basically the experiences a user had when using the system. Surveys are studies that produced descriptive statistics or collected data from user feedback. We consider experiments that do not compute a statistical significance between groups as surveys.

The section is organized into the following subsections:

1) Experimental and survey evaluations
2) Qualitative evaluations
3) Survey evaluations

1) Experimental and survey evaluations

This section contains a review of the publications which evaluated program visualizations either via experiment or survey. The two types of evaluations are combined in one section because many experimental studies are followed-up on by feedback questionnaires. Keeping such studies together in a single section should improve readability of our paper.

EDPVE

EDPVE was used in an experiment to compare the effectiveness of static and dynamic program visualizations. A complementary system called example-based static program visualization environment (ESPVE) was used for comparison. The systems supported only a flowchart view, variables and

source code, which could not be animated. In fact, the user could not have any interaction with the program. Also, as opposed to EDPVE, the system does not provide information about the current instruction being executed. Both systems were used as support for traditional teaching methods (Tekdal, 2013).

The authors used a test that included the following subject areas: variables, control statements, loops and arrays. They report a statistically significant difference in favor of using dynamic program visualizations on the overall test. Additionally, they divided the test items into three subtests: control statements, loop statements, and array statements. For each subcategory, they report a statistically significant difference favoring dynamic program visualizations (Tekdal, 2013).

A delayed was used to assess student retention. The results of the posttest and delayed test showed no statistically significant difference for either group. The result indicates that the type of visualization does not affect retention. However, it is unclear whether the delayed test was the same as posttest, which seems to be the case. If this is correct, then there is a possibility that students have remembered (some of) the answers (Tekdal, 2013).

It is unclear whether the level of content ownership might have affected the results. The authors note that the systems were used merely as support. However, we need to ask ourselves if the group using static visualization would have interacted more with the system if it supported own cases, or a higher level on the content ownership dimension. A higher content ownership dimension might have prompted students to explore the static visualization more. One might argue that providing own cases to a static visualization is irrelevant. Given that the variables were visualized in both systems, we believe it might lead to a higher degree of interaction.

JAD

JAD was used in a study to compare the effectiveness of its visual debugger for learning programming in a basic Java course. The study compared the performance of DHI and non-DHI students to fix bugs in two Java classes. Three performance metrics were used: time to complete task, completion of all tasks, and number of times the student asked the instructor for help. The p-values showed no statistically significant difference between DHI and non-DHI students on all metrics. However, the study was conducted on only 10 students (5 DHI and 5 non-DHI) (Nascimento et al., 2017).

JAD was qualitatively evaluated by DHI and non-DHI students. However, we consider this a survey, since the theoretical framework and detailed results for the qualitative evaluation were not provided in the publication. In general, the study reports that DHI students found the tool more appropriate than other tools, and was intuitive, simple, and helped with debugging (Nascimento et al., 2017).

JaguarCode

JaguarCode's (formerly JavelinaCode) effectiveness towards helping students comprehend two given projects was evaluated in two controlled experiments. Both experiments had the same two projects, a single control group and an experimental group. Both experiments evaluated students' response times and correctness of solutions to given questions. The time it took to answer each question was recorded (J. Yang, Lee, & Chang, 2017; Jeong-sug Yang, 2016).

In each experiment, the control group had no visualizations, and the experimental group used JaguarCode (JavelinaCode at the time). In both experiments and for both projects, students' in the experimental group took longer to answer the questions. A statistically significant difference was found in the harder of the two projects. Regarding the correctness of solutions, in all experiments and projects, the results favored the experimental group. Statistically significant difference in the first experiment was found for the easier project, and for both projects in the second experiment. The authors concluded that using JaguarCode improved student performance on questions regarding tracing and program understanding. The higher response time for the harder questions might be the result of students using the visualization to improve the accuracy of their answers. The experiment might be considered a comparison of different engagement levels: no viewing vs viewing for program understanding and tracing (J. Yang, Lee, & Chang, 2017).

In addition to the experiment itself, the authors evaluated the system's usability and visualization with two questionnaires. Results of ow that students were generally satisfied with both static and dynamic visualizations provided by JaguarCode. Some participants also suggested certain improvements, such as multi-language support, increasing the execution speed and better feedback on errors (J. Yang, Lee, & Chang, 2017; Jeong-sug Yang, 2016).

An experiment by Yang et al. (2017) investigated if JaguarCode's run-time visualization helped students to understand OO concepts and if generated UML diagrams assisted users with interpreting their program's behavior. The published article reports only on the feedback collected via questionnaire. The study reported positive feedback on the system's usability and visualizations. In general, most participants agreed that the generated UML diagrams helped them with program comprehension. Most of them reported that the class diagram helped them the most with understanding the program, followed by the sequence diagram. The participants also reported that JaguarCode is useful for beginners to learn java programs and the visual diagrams were helpful (J. Yang, Lee, Gandhi, et al., 2017).

Codeeasy (Jeliot 3)

Eranki & Moudgalya (2013) used Codeeasy, a tool built on Jeliot 3 to investigate the effect of a PV system in spoken tutorial workshops. The study included four groups in total. Two groups, one control and one experimental, learnt Java, and the others learnt C++. All groups watched spoken tutorials, while only experimental groups used the Codeeasy. The authors found a statistically significant difference for learning programming competencies in favor of PVs. Program comprehension and debugging skills were significantly improved for the experimental group. The same participants also demonstrated better performance concept-wise.

Jeliot ConAn

Moreno et al. (2014) carried out an experiment to compare the effectiveness of *Jeliot ConAn* (conflictive animations, experimental group) and *Jeliot 3* (normal animations, control group). The study evaluated conflictive animations related to *function calls*.

Results of the study showed that Jeliot ConAn had an impact on students, and they slightly improved their score in the post-test relative to the pre-test. However, no statistically significant difference was found between the control and

experimental groups. The authors believe that this might be due to the small number of participants (N = 18) (Moreno et al., 2014). Apart from evaluating conflictive animations, an impact on study learning may have been caused by the difference in engagement levels of the two systems, *responding* vs. *controlled viewing*.

An additional graphical questionnaire was used to evaluate students understanding of animations and concepts in them. A correlation between the graphical questionnaire and students prior programming knowledge was found for both groups. In the graphical questionnaire, the control group obtained a higher average score (control group = 6.82, experimental group = 4.82). The authors reported that some explanations from the control group indicated that they had a better understanding of the visualized concepts than the experimental group (Moreno et al., 2014).

Moreno et al. (2014) also collected student feedback about Jeliot ConAn and Jeliot 3. In general, twice as more students in the control group reported that Jeliot 3 helped them understand Java programs than in the experimental group. Both tools were not hard to use, while the control group wished they used the Jeliot 3 in more debugging exercises.

ObjectVsualizer

ObjectVsualizer was evaluated in two experiments, one of which focused on debugging, and the other on extending functionality tasks. Both experiments featured a control (no visualization) and experimental group (visualization). In both experiments, students that used the visualization obtained higher scores in average. However, only the first experiment showed statistical significance between the groups. These results indicate that the use of visualization might not have a significant impact on understanding simpler programs (Shin, 2018). This study may also be considered as no *viewing* vs. *viewing* in the engagement taxonomy.

Online Python Tutor

Karnalim & Ayub (2017a) carried out an experiment on *Online* Python *Tutor*'s effectiveness in an introductory programming course. They included two classes in the study which alternately used OPT, except for the fifth (both did not use OPT) and sixth week (both used OPT). The study included both quiz and questionnaire. The results of the questionnaire showed that OPT is a promising PV system for learning introductory programming and students would like to use it more laboratory sessions. Quiz and questionnaire results both showed that OPT has a positive impact on doing basic programming sub-task, such as understanding program flow. For more advanced topics, which were also evaluated with quiz and questionnaire, the visualization had a positive impact on learning functions, while it did not have such an impact on arrays.

Karnalim & Ayub (2017b) also conducted a survey to collect feedback on students' perspectives of OPT. The participants were divided into two groups that used OPT for 14 weeks alternately, one group used it in odd weeks, and the other in even weeks. The feedback is generally positive. Students believe that OPT can help in finding errors and understand how their code works. However, students need time to adapt to using the system and slow internet connection may discourage its use.

Karnalim & Ayub (2018) also carried out a quasi-experiment to evaluate Online Python Tutor's effectiveness for learning data structures. The experiment was carried out in 14 lecture weeks, with one group being the experimental group for odd weeks, and the other for even weeks. The control group for that week did not use visualizations. The authors carried out comparisons between intervened and non-intervened sessions within the same group. Results showed statistically significant difference, for one group a positive correlation, and for one negative. The authors assume that this discrepancy occurred because OPT's UI is not intuitive. They assume that the group with the positive correlation had prior experience with OPT. Authors also compared the corresponding sessions of the same week between groups. Statistically significant difference was shown in the first week, which favored the control group, and the twelfth, which favored the intervened group. These results are probably related to the adaptation time required to get used to OPT's UI and visualization. The authors conclude that OPT might be an effective learning tool if the students have a chance to use it.

Thayer, Guo, & Reinecke (2018) studied the correlation between the users back-stepping through the visualization and their cultural levels of self-centered learning. They took into account the *Power Distance Index* (PDI) and *Conservation*, which measure instructor-directed learning. A higher PDI indicates that a country's educational system is centered around the authority ("teacher-centered education"), and a lower that it is centered around the student ("student-centered education"). Conservation is a measure of how much the student values tradition. They carried out two studies and showed that OPT did not benefit students from all cultures equally. Their first study, that focused on country-level PDI, showed that students from cultures that have a lower PDI will take more back-steps. Their second study investigated how culture affects back-steps, and personal values and back-steps affect debugging success. The study found 1) only marginal negative correlation with Conservation, 2) the number of back-steps was negatively correlated with debugging success, and 3) for instructor-centered learners, many back-steps meant lower debugging success, and for student-centered learners the correlation was lower.

PandionJ

PandionJ was used in a course offering of 2017/2018 for 12 weeks. The pass-rate of the course was compared to the pass-rate of three prior years when AguiaJ was used ("AguiaJ," n.d.). The results showed statistically significant improvements in course pass-rate compared to each of the three prior years (Santos, 2018).

PITON

Elvina et al. (2018) evaluated PITON in two studies, one with lecturer assistants and one with students. Both lecturer assistants and students provided feedback. Feedback from the first study was mostly positive. Lecturer assistants believe that the step-by-step execution can be helpful for students, and that the error message descriptions are easier to understand than standard ones. A recognized shortcoming of PITON is that it does not visualize object variables.

The second study investigated the effectiveness of PITON compared to PyCharm+OPT scenario. The authors concluded that PITON is more beneficial to use in laboratory sessions because it is impractical for students to switch between PyCharm and OPT. Additionally, this may have resulted in comparing PyCharm with PITON, which led to a statistically significant difference in favor of PyCharm+OPT for the topic of functions. A survey of students showed that they preferred

PITON over PyCharm+OPT and that its implemented features are helpful (Elvina et al., 2018).

DS-PITON was evaluated with several quasi-experiments to measure its effectiveness for learning data structures compared to traditional textbook learning. Two quasi-experiments included students that have passed the Basic Algorithm and Data Structures course with a grade of C or higher (moderate-paced students), and three quasi-experiments included students still enrolled in the course with a mid-term score below C. The comparison took into account the score and time required to complete an assessment. Results of the quasi-experiments showed a statistically significant improvement in favor of DS-PITON for both moderate-paced and slow-paced students. Regarding time efficiency, a statistically significant improvement was observed for moderate-paced students. For slower-paced students, DS-PITON did not increase time-efficiency, and in one experiment, it showed a statistically significant increase in completion time. The feedback from the survey collected information on potential improvements to DS-PITON and showed that students believe in its effectiveness for learning data structure materials (Nathasya et al., 2019).

PlanAni

PlanAni was used in an experiment to study the effect of visualization on understanding the roles of variables. The programming language taught was C. The results of comparing program construction between the control and experiment groups yielded no significant results. Hence, the authors compared them based on the SOLO taxonomy (Lister, Simon, Thompson, Whalley, & Prasad, 2006). A statistically significant result was obtained on the SOLO levels for program construction. The difference favored teaching the roles of variables visually (experimental group) (Shi, Min, & Zhang, 2017).

A feedback survey showed that students in the experimental group had a higher rate of approval about the roles of variables. However, both groups were not satisfied with their ability to construct programs (Shi et al., 2017).

PVC

Ishizue et al. (2018) carired out an experiment in which they compared PVC, SeeC and no visualization. The PVC had the largest percentage of correct answers, with a statistically significant difference with regard to SeeC. The group that used SeeC had the lowest score. However, there is no report that a pre-test was used to check for difference between the groups. Results from the questionnaire show that most of the students find PVC useful for introductory programming and more accessible than other PV tools.

Kumalija et al. (2019) investigated students' perception of SunLab's features. Given that smartphones have a significantly smaller screen than other devices, the results were promising. The animations were fairly visible and text editor was moderately difficult to use. Students' also agreed that SunLab helped them understand programming concepts, with loops and functions obtaining the lowest score.

TEDViT

TEDViT was evaluated in classroom practices for learning sorting and search algorithms, and the difficulty of constructing T-Rule sets (Ihara et al., 2017; Yamashita et al., 2015, 2016). Yamashita et al. (2017) found that software engineers without much experience with C found TEDViT's support for learning pointers valuable and were generally satisfied with practice sessions involving TEDViT.

TEDViT (experimental group) was compared with ANIMAL (control group) in an experimental setting to evaluate its effectiveness for learning recursive functions. The results show that the experimental group achieved much better scores as well as a higher score increase ratio. Additionally, TEDViT obtained higher scores for almost all questions in a feedback questionnaire (Yamamoto et al., 2017).

WinHIPE

WinHIPE was used to compare the effectiveness of different engagement taxonomies on student learning: *no viewing* and *viewing*, *constructing* and *no-viewing*, and *constructing* and *viewing*. The experiment compared student scores based on three points of view. The global point of view, which gave a single point for each student showed no statistically significant difference. At the Bloom's level point of view, five scores were assigned to each student, each score for the questions related to a certain level of Bloom's taxonomy. Students that were engaged with WinHIPE obtained a statistically significant higher score at the analysis and synthesis levels. From the topic point of view, scores were calculated for each topic. At the topic level, the viewing group outperform both other groups on recursive functions, and the viewing and constructing groups outperformed the no-viewing group with a statistically significant difference. With regard to long-term results of the study, the viewing and constructing groups had a statistically significant higher pass rate. The constructing group also had a statistically significant reduction in drop-out rate. Participants were satisfied with the use of WinHIPE (Urquiza-Fuentes & Velázquez-Iturbide, 2013).

2) Qualitative evaluations

This section contains reviews on the qualitative evaluations of PV visualization features. We do not consider feedback from questionnaires and surveys as qualitative evaluation.

Omnicode

Omnicode was evaluated on a group of students who were asked to solve three introductory programming problems in Python using the system. A questionnaire was administered, and a debriefing interview was conducted. Although the results of the questionnaire were encouraging, it identified a potential problem with visual overload. Interview results report that students believe Omnicode is useful for helpful for the formation of correct mental models and for providing teaching and explanatory assistance. Students successfully used the visualization to debug their code and improve their mental models. The authors note that students used the scatterplot view provided by the system when explaining how the code works. An issue with Omnicode is the large amount of visualized data which leads to visual overload. Students suggested some improvements on how to reduce the number of scatterplots (Kang & Guo, 2017).

UUhistle

UUhistle was used in a phenomenography study to understand the way learners experience VPS and what they can learn from VPS. The authors conducted semi-structured interviews that revolved around VPS exercises. Results report six logically connected categories of various VPS perceptions. VPS is perceived as the manipulation of visual components and supported actions in the simplest category. In the richest

category, VPS is perceived as improving programming skills through understanding implementation concepts and what the computer does. The other four categories are between and form two branches of two categories. Both branches extend the simplest category and are extended by the richest. One branch emphasizes the operations of the computer, while the other emphasizes VPS as a platform for studying code examples and present concepts (Sorva, Lönnberg, & Malmi, 2013).

VIP

Isohanni & Knobelsdorf (2013) studied how learners engaged with the visualization tool *VIP*. Their research was not based on any existing engagement taxonomy since this might lead, as the authors reason, to ignoring certain forms of engagement. After grouping user activities into several layers of abstractions, the authors recognized four groups that describe "levels of visualization engagement": *no need for VIP*, *using VIP fully*, *using VIP partially*, and *unfinished use of VIP*. These groups were further related to the internalization concept of activity theory. The authors also recognized four different phases of using VIP, which can be used to describe a learner's certain advancement in the use of VIP. The phases are *introductory use of VIP*, *progressive use of VIP*, *routine use of VIP* and *creative use of VIP*.

Ville

Ville was evaluated as a collaborative learning platform in different teaching scenarios (Laakso et al., 2018).

3) Surveys

Online Python Tutor (Codechella)

Guo et al. (2015) evaluated how users interacted within the Codechella system. Here we focus only on the interaction with the visualization provided by OPT. Most of the actions with the visualization included stepping through the visualization. The average length of the programs was not reported in the study, but given that users asked help from others, we may assume that they had at least around a dozen lines of code. Although, it is not considered part of the visual representations, chatting is also included and is the second most recorded activity. Again, this makes sense since users needed to communicate explanations of code. It would be interesting to see which visualizations of concepts, algorithms or data structures correlate the most with the number of exchanged chat messages. Running and editing were reported as the least recorded activities.

TM Visualization

Al-Fedaghi & Alrashed (2014) carried out two experiments measuring student understanding of C++ programs using *TM visual representations*. The first experiment had a single group that used TM representations, while the second had two. Both experiments included one group that had no visualization. In both experiments, the groups that used FM representations had better scores in average. The second experiment also showed that the variance in scores is higher in the group without TM visualization. However, scheduling and registration limitations prevented more reliable analysis. In the context of engagement taxonomies, this one could be considered as a comparison of *no viewing vs viewing*.

jGrasp

jGrasp's viewers were investigated in several studies. They mostly investigated the systems' effectiveness towards learning algorithms and data structures. However, based on these, the authors refined the viewers for use in other courses, such as CS1. A survey of CS1 and CS2 educators from various faculties was carried out in 2011, which showed that the educators believe that jGrasp could have a positive impact on learning. These educators attended a workshop in visualization for CS1 and CS2. A year later, the authors surveyed students that used jGrasp in a CS2 course, which again showed a positive attitude towards jGrasp's canvas (Cross et al., 2014).

The authors also conducted a survey on students enrolled in a CS1 course in 2013. Students were required to complete an in-lab activity that focused on binary and linear search in Java. Authors report that the students' felt that jGrasp had a positive impact on their learning (Cross et al., 2014).

LISN

LISN's visualization and language-independence aspects were evaluated on a group of lecturer assistants. The results were fairly positive, with some of the participants pointing out that additional details for variables are required. Incorporating new programming language support into LISN was considered simple (Sulistiani & Karnalim, 2018).

PROVIT

Yu Yan et al. (2014) evaluated PROVIT with a group of high-school students. About a half of the students said that they can understand C programs, and very few stated that they can write them. PROVIT-CI's features were evaluted by students and instructors (YAN et al., 2018). Results showed that the system was highly accepted by students, and instructors found it helpful during classroom lecture instructions.

Thonny

Annamaa collected Thonny activity logs from 44 students. While replaying the logs, the authors noticed some interesting working patterns. The authors report that students found Thonny helpful in dbugging their programs and they liked that the shell and editor were in the same window (Annamaa, 2015).

Virtual-C IDE

Virtual-C IDE was evaluated by comparing three years of students' scores and failure rates. The first year saw students use commercial IDE, the second used Virtual-C in lectures and a commercial IDE for programming assignments, and only Virtual-C IDE was used in the third year. The failure rate dropped considerably in the final year, which also saw an increase in student scores. The integrated functional tests required students to fix their errors which lead to students spending more time on their programming assignments (D. Pawelczak & Baumann, 2014).

Pawelczak (2016) found that a class which used the testing framework had a lower failure rate compared to the one that did not. The transparency of the tests encouraged students to more willingly comprehend their errors and lead to less time spent on debugging programs.

VIII. Visual programming languages

Visual programming languages can be taught of as a subcategory of program visualization (Price et al., 1993). The visual representation of concepts provided by a VPL may be taught of as components of a visualization. Hence, novices must engage with the components provided in order to construct a working program. The result is a program itself as

well as a visual representation of its structure. Some VPLs, e.g. Scratch, have a theatre view which can animate the behavior of the program's concepts, or a subset of them. Some flowchart VPLs allow the user to step through the execution of a program (S. Xinogalos, 2013).

A recent literature review on VPLs investigated the effects of using visual languages in introductory programming courses. The authors found that VPLs can have a positive effect on retention rates and interest in programming when used with the right age group. It was also reported that the choice of first programming language for learning programming is not as important as the teaching methodology. Preferred programming languages include Python and Java as text-based languages, and Scratch as a VPL. In a course setting, it may be beneficial to combine the use of a VPL and a textual programming language (Noone & Mooney, 2018).

Many VPLs are intended to assist novices in learning to program. One of the most well-known examples is Scratch (Maloney et al., 2010), which provides block-based representations of programming concepts. Each block has a jigsaw-like shape which significantly simplifies syntax. VPLs are also often developed to assist non-programmers with some workflow. For example, LabView is a dataflow visual programming language (DFVPL) whose goal is to assist researches create applications (Alireza Kavianpour, 2014).

There are different ways for classifying VPLs. Burnett & Baker (1993) proposed a classification system for VPL literature. They propose the classification of visual languages based on the programming paradigm or visual representation. Three main representations are identified: diagrammatic, iconic and static pictorial sequences. Myers (1990) also proposed a classification based on the visual representation of a language. The classification is much finer-grained. For example, the classification includes directed graphs and data flow as two categories, which are both diagrammatic representations.

Zhang (2007) classifies visual programming languages as diagrammatic, icon, and form-based. Diagrammatic languages are composed of nodes that are connected with edges. Dataflow programming languages are a type of diagrammatic programming languages. Iconic programming languages use symbolic representations for variables, data types and control (S. Xinogalos, 2013). Icons are attached to one another and can be combined into more complex concepts (Bácsi & Mezei, 2019). An example of an iconic programming languages is Lego Wedo ("LEGO® Education WeDo 2.0 Core Set.," n.d.). Based on their definition and similarity to Lego Wedo, Scratch, which is often referred to as a block-based programming language, would also be considered an icon programming language. Not all icon programming languages are block-based. Spreadsheets are an example of form-based programming languages (Zhang, 2007).

Bácsi & Mezei (2019) note that visual languages can also be categorized based on the relation type. Connection-based are, just like diagrammatic languages, represented with nodes and edges. Containment-based programming languages are those whose elements are embedded into other elements and combined to form visual sentences, e.g. Scratch.

Xue et al. (2017) proposed a classification based on the execution model of visual programming languages. They identified control flow, data flow, state transition and constraint-based languages.

Erwig et al. (2017) proposed an ontology for visual languages which characterizes specific languages through *profiles*. A profile of a visual language is the combination of essential and derived tags which capture the language's aspects. The authors identified four essential tags that characterize a languages visual representation: graph, partition, icon, text. This ontology does not make a distinction between VPLs and any other visual languages.

Despite the many categorizations and taxonomies for VPLs, most of them fall into block-based and dataflow visual programming languages (Mason & Dave, 2017). Block-based programming language natively support programming in the imperative programming paradigm. DFVPLs' programming paradigm corresponds to functional programming. In the next section we will discuss DFVPLs in more detail.

A. Dataflow visual programming languages

Programs in DFVPLs are constructed by connecting nodes with edges. The resulting program is a directed graph. Different nodes may require different types and numbers of inputs and may produce one or more outputs. The execution starts from activation nodes that usually provide some data which flows downstream until it reaches a node without any output arcs (Johnston, Hanna, & Millar, 2004). The nodes transform the data as it flows through it. This transformation corresponds to the way functions chain together and transform data in functional programming languages.

Nodes in a dataflow graph can be taught of as visual components of a notional machine. These components may operate at different levels of abstraction. For example, a dataflow visual language may provide low level instructions as nodes, such as basic mathematical operations. On a higher abstraction level, the nodes could provide functions implemented in another language, e.g. Python or Java (Johnston et al., 2004). Hence, depending on the target audience, dataflow languages may provide different components for constructing programs.

There are two main execution modes in the dataflow model: *data-driven* and *demand-driven*. In the data-driven approach, a node executes as soon as data is available on all of its inputs. In the demand-driven approach, a node activates when it receives a request for its data. Requests are propagated upwards through input edges, and data is sent downwards through output nodes. In both cases, a node places the result of its execution on its output arcs (Hils, 1992; Johnston et al., 2004).

DFVPLs provide several key aspects which may help in understanding program execution. The graph representation of a program makes the relationship between program components explicit. Users can visually explore data for a more concrete programming experience. DFVPLs also provide visual feedback on different levels of liveliness (Johnston et al., 2004). The four levels of liveliness described in literature build upon one-another. A higher level includes all of the lower levels. On the *informative* level, the system is used only to document the program or assist with understanding it. The *informative and significant* level allows the user to execute the constructed program. The third level, *informative, significant and responsive*, encompasses systems which re-execute the graph whenever the user changes

anything, e.g. modifying data. Finally, the fourth level also assumes that the system can process data streams while the user is editing the program (Hils, 1992; Tanimoto, 1990).

An advantage of DFVPLs is that it provides explicit specification for task-level parallelism. All nodes whose data is available at the same time step may be executed in parallel (Hong, Oh, & Ha, 2017; Johnston et al., 2004).

B. DFVPL issues

A few notable issues with DFVPLs that have been recognized decades ago still seem to be open. These include visual representation and iteration and control constructs (Hils, 1992; Johnston et al., 2004; Sousa, 2012).

1) Visual representation issues

Visual representation issues arise when programs are constructed from a large number of nodes. Besides the program not fitting the screen, a visually complex program may be more difficult to comprehend than its textual counterpart. A solution that naturally arises is to provide hierarchical grouping of nodes. Complex groups of nodes are then represented as a single node (Sousa, 2012).

2) Control and iterative construct issues

Common control and iterative constructs found in textual programming languages control the flow of the program. However, nodes in a dataflow program transform data and control which nodes it flows to. Hence, it is difficult to transfer these constructs into a dataflow environment.

Instead of common control structures, DFVPLs usually provide merge and switch blocks. The merge block takes two inputs and a Boolean "control" signal. The control signal determines which input will be forwarded as output. The switch takes a single input and a Boolean "control" signal. However, the switch block has two outputs. The Boolean signal determines to which output the data will forwarded (Johnston et al., 2004).

Iteration constructs are much more difficult to represent. Although different DFVPLs proposed several different solutions, it remains an open issue. DFVPLs usually provide different nodes for different iteration constructs. Some DFVPLs like *Show and Tell* and *LabView* use nodes that encompasses other nodes which represent the body of the loop (Johnston et al., 2004). *VIPERS* provides an iteration construct that does not encompass other nodes. Rather, different output edges are used to connect the body and continuation once the loop exits. It also requires a feedback control signal to determine when the execution of the next iteration should take place. The two examples provided represent two main approaches for representing iteration constructs: the first avoid cycles in a program graph, and the second uses them (Mosconi & Porta, 2000).

A key issue that makes iterative construct difficult to represent is that they are required to mutate some state. This requirement infringes the single assignment rule natural for dataflow programs (Mosconi & Porta, 2000).

C. Actors and dataflow

The actor model is a message-passing concurrency model first introduced as a formalism in artificial intelligence (Čolak & Čuvić, 2019; Hewitt, Bishop, & Steiger, 1973). Basic primitives in the actor model are *actors*, i.e. computational agents, which are autonomous and operate asynchronously and concurrently. Actors can communicate with each other by sending messages and exhibit some behavior when a message is received. After receiving a message, an actor can carry out computations, change its behavior, create new actors, reply to a message or update some local state (Agha, 1986).

Actors have a mailbox for storing messages, which are commonly stored in a queue. Each actor in the actor model sequentially processes the messages from the mailbox in a FIFO fashion. The programmer does not need to worry about thread management or locking ("Actors • Akka Documentation," n.d.). These aspects of the actor model simplify reasoning about the execution of concurrent programs.

In the dataflow model, a node may be taught of as an actor. An actor is viewed as a processing node which exhibits some behavior when a message is received. Messages exchanged between actors can be taught of as communication channels, or edges (Sousa, 2012). However, the relationship between the models seems to require that the graph is a type of Kahn processing network. In Kahn processing networks, communication is achieved through unidirectional FIFO queues, where writes are nonblocking and reads are blocking operations (Lee & Parks, 1995).

IX. Visualizations for Teaching

In this section we introduce two visualization systems that we have developed for teaching and learning. The first system which we will describe is a DFVPL for learning introductory and functional programming that we call PARVIS. The second system is AkkaVisual, which is a PV system for visualizing actor programs.

A. PARVIS for introductory programming

PARVIS is a web based DFVPL initially developed for teaching introductory programming. The system combines certain aspects found in PV systems with the DFVPL.

Controlled viewing that is implemented in many PV systems allows users to control the execution of the program in a step-by-step fashion and highlights the currently executing instruction. PARVIS similarly allows the user to step through the execution of the program graph. Programs are parsed and executed in the background to determine the number of steps needed to completely execute the program. The parser uses a modified BFS algorithm to traverse the program graph. The result of the parser is a key-value data structure. The key is a step in the execution and the value contains all of the nodes that will be executed at that step (Aglić Čuvić, 2018).

In the case where two nodes are mutually independent, it is problematic to determine which node should be highlighted first. Therefore, all of them are highlighted simultaneously. The step through can be taught of as visiting the nodes in a fashion similar to breadth-first search (BFS).

It is possible to have a situation in which node A expects multiple inputs and one, from node B, becomes available at an earlier step than the others. In this situation, node B will be highlighted for each successive step until all of the other inputs to A become available.

PARVIS provides different types of variable nodes for numbers, strings, Boolean values and arrays. For each value stored in a variable node, there is a textbox that prints it out. Users can edit the value if the node does not have any inputs

connected. Therefore, the user can see the state of the program at each step.

We wanted to be able to demonstrate Python programs with PARVIS. Hence, the system provides print and input nodes. Just as Python's print instruction can take an arbitrary number of string arguments, the print node can take an arbitrary number of inputs. The values of the inputs are printed in a designated area in the order of their position on the canvas along the Y-axis. When the input node is executed, a modal window opens for the user to enter the value. These behaviors are similar to how PV systems handle input and print instructions.

Control flow nodes include if, elif, and else. Because we wanted the nodes to behave similarly to Python instructions, their output edges do not carry data. They just provide a way to add nodes to the program that will execute after the control flow node. However, the output edges of these nodes must be connected to activation nodes, such as variables. The system color codes which branch of the program will be executed. Merge and distributor block are also provided.

An example program is given in Fig. 2. The program in the image prints out a message depending whether the input to the if node is true or false. The highlighted print node prints the value in a special output area. The output area is also highlighted to signalize that a print occurred.

PARVIS also provides a node for Python's for loop. The node is actually a two-part node that consists of a head and body node. The nodes are connected via an edge directed from the head to the body, which is not considered an input arc. The body node encompasses all of the nodes that will be executed in each iteration. When a user connects an encompassed node's output to the loop body, an output and corresponding input port are generated. The value of the node then re-enters the body node at the corresponding input port.

The system also supports a high level of liveliness. The program is re-evaluated up to the current step whenever the user edits the program or changes a variable value. Hence, it provides instant visual feedback about the effect of user actions on the program.

We used PARVIS in an experimental setting with a group of students enrolled in the application of computers course at the Faculty of Chemical Technologies. Students were studying the basics of Python programming: variables, control constructs and the for loop. The system was used to demonstrate Python programs and students were urged to use the system. The other group was given static flowchart representations of the examples.

From our experience, the students used PARVIS to some extent to execute the given examples. However, they did not like the system. The visual representations were too complex for the short and simple programs that they were given. We have not yet statistically analyzed the collected data.

We concluded that the low granularity of PARVIS and complex visual representations were not appropriate for teaching simple Python programs.

B. *PARVIS for functional programming*

We later extended PARVIS with additional nodes to support teaching and learning functional programming. Since the programming paradigm of dataflow languages correspond to the functional programming paradigm, the extension

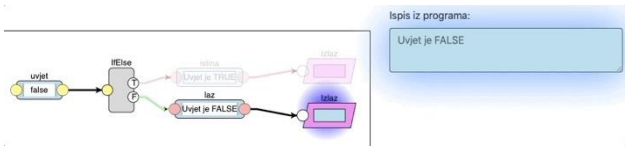

Fig. 2 An example program in Parvis

seemed natural. As functions transform input data, so do the nodes in a dataflow language. Hence, we added nodes for the map, filter and reduce functions among others.

The input to the introduced higher order functions needs to be an array. Map and filter produce new arrays, and the output type of the reduce depends on the accumulator value which the user needs to enter. When the execution step reaches an array node, the parser uses a modified DFS algorithm to determine a subgraph that starts from the array and ends either with the end of the program or another variable type node. Once finished, new stepping controls become visible on the interface. These allow the user to step through the elements in the array. At each step the current array element and subgraph are highlighted. If the subgraph ends with a variable type, then the transformation of values is visible for each step. For example, if the subgraph is array-map-map-array2, the result of passing the element through the two maps is added to array2. The next step will add the second element and so on. The added functionality allows users to inspect how an array is transformed per element when data flows through a chain of functions.

One of the limitations of the system is that higher-order function nodes do not provide a way to define function parameters. Instead, the user needs to write a function in JavaScript.

PARVIS was used in eight lab sessions in which students were taught functional programming. The lab sessions usually included up to 7 students. We gave the students examples in both PARVIS and JavaScript and assignments which the students were required to program in PARVIS. They were reluctant to use the system at first but accepted it after the first or second session. The second to final session included a preliminary test with three assignments in JavaScript and three in PARVIS. For the final session we administered a test with four assignments, two of which had to be programmed in JavaScript and two in PARVIS. JavaScript and PARVIS assignments were similar and had a comparable difficulty. An additional assignment was given for which students could choose the implementation language.

Because the system was still in its prototype stage, it could not detect syntax and semantic errors in parameter functions. Therefore, if an error occurred inside the parameter functions, students were often required to reload the DFVPL web page. The problem caused some difficulties with the system's usability.

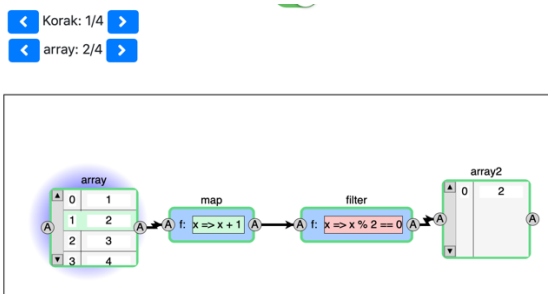

Fig. 3 An example of executing arrays element-wise

Since students are not used to think functionally, they may start program in an imperative style. One of the benefits of the DFVPL was that it forced students to use functional programming. One student commented that the "system directs towards the functional programming paradigm… here's five nodes and figure it out". We still observed students using the imperative programming paradigm for some assignments or parameter functions.

Seven students took the final test during the eight session. The average score of the assignments in JavaScript (avg=2.857) was lower than those in PARVIS (avg=4.286). An additional three students took the test at a later time, their average score for JavaScript assignments (avg=4.333) was slightly higher than the PARVIS average score (avg=4). Combining all student scores give an average that favors PARVIS (avg=4.2) over JavaScript (avg=3.3).

The scores show that using PARVIS did not have a negative effect on learning functional programming. Because of the small number of students and lack of a structured test, more rigorous statistical analysis is not possible. However, the observed scores are encouraging. They indicate that PARVIS could assist students with learning functional programming. However, additional work on the system is needed to eliminate possible bugs.

C. AkkaVisual

AkkaVisual is a PV system for visualizing actor programs. It visualizes actors in a program and their communication channels. Hence, it corresponds to a visualization of a message-passing notional machine (Sorva, 2013). The system currently offers viewing the visualization while the program is executing. Actors are represented as nodes in a graph and the message channels are edges. The system also displays a timeline of message sent events. Messages are order according to the vector clock algorithm (Čolak & Čuvić, 2019).

AkkaVisual is a web-based application that collects JSON data published to an exposed API by the running program. The data will typical contain information about the sender and receiver actors, such as their name and type. It will also contain certain details about the message, such as the type and value contents. Once a JSON object is received, the server side of the application uses SignalR to push it to the client-side (Čolak & Čuvić, 2019; "Real-time ASP.NET with SignalR | .NET," n.d.).

Any standard HTTP POST request can send data to the API. The separation of the visualization from the program itself allows AkkaVisual to support visualizing OO and agent-based programs. The executing programs simply need to publish the required data in the correct format (Čolak & Čuvić, 2019).

At our Faculty, we use C# and Akka.NET to teach the actor model. For convenience, we implemented a custom actor mailbox intercepts received messages and publishes a copy of them to AkkaVisual. The mailbox can be easily added to a student project and actors can be configured to use it via config file. Fig. 4 shows an overview of this process (Čolak & Čuvić, 2019).

AkkaVisual was evaluated in a pilot study with a group of students. Unfortunately, students did not have an opportunity to use the system. Therefore, we demonstrated its functionality in front of them. Data was collected using a questionnaire which consisted of seven 5-point Likert-scale questions and an open question. The Likert-scale questions asked students to rate how useful they thought the system and some of its existing, and planned features, are for learning actor programming. The open question was included so that students could share their thoughts about the system. Students that did not take the course were excluded from the data analysis process. A total of 19 students filled out the questionnaire, and 11 answered the open question. Descriptive statistics were used to analyze the quantitative data and content analysis for the qualitative (Čolak & Čuvić, 2019).

Results of the descriptive analysis are given in Table 6. All of the mean and median scores are high, which indicates that students believe that the system and its features could be useful when learning actor programming. We briefly comment on some of them.

The highest scored feature is saving and replying the visualization at will (avg=4.53). We plan to implement this feature in the future. Possible explanations for this high score include: 1) wanting to share the visualization without sharing the program and 2) saving time. Regarding the first explanation, a saved replay would allow students to help their peers without sending them the solution of an assignment. Students that struggle with an assignment could compare their visualization to that of a correct solution. The second explanation is that students do not want to re-run the application multiple times because it might have long-running computations or require multiple inputs that need to be entered manually (Čolak & Čuvić, 2019).

The second highest scored feature (avg=4.37) is the information provided about the actors and the messages they exchange. An actor system may have different types of actors that handle multiple types of messages. Students often find it difficult to understand which communication channels are formed and to find bugs when they send a message of the wrong type. Hence, providing information about the actors and their message exchanges might alleviate some of the difficulties (Čolak & Čuvić, 2019).

Students also gave high scores to other existing and planned features (avg >= 4.21). They also believe that the system is adapted for students learning about actors for the first time(avg=4.05) (Čolak & Čuvić, 2019).

Using content analysis, we derived eight codes from the responses to the open question. These eight codes are grouped into three categories: *positive about the tool*, *the tool needs improvements* and *tool isn't useful*. The category positive about the tool contains cods that indicate that AkkaVisual could be useful for teaching the actor model. The second category is related to improvements and new features required. Finally, the third category contains codes for answers that believe the tool will not help with learning the actor model. Fig. 5 shows an overview of the categories and codes derived with content analysis (Čolak & Čuvić, 2019).

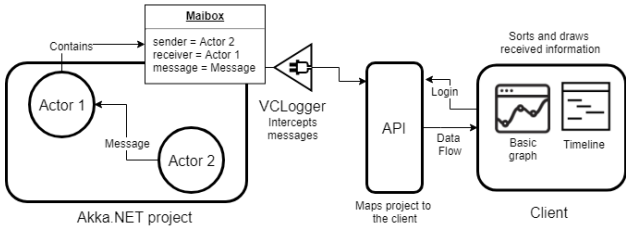

Fig. 4 An overview of how data is collected and displayed in AkkaVisual

TABLE 6 DESCRIPTIVE RESULTS FOR AKKAVISUAL FEATURES (PILOT STUDY)

#	Question	Score				
		Avg	*Std. deviation*	*Mode*	*Median*	*N*
1	The tool is adapted for first-year graduate students learning about the actor model for the first time.	4.05	0.911	4	4	19
2	I would like to see a timeline of the execution of each thread or process where the exact time of sending and receiving each message is displayed.	4.21	0.976	5	5	19
3	The replay feature seems useful.	4.16	0.898	5	4	19
4	The timeline seems to be useful in visualising the concurrency of the system.	**4.32**	0.749	5	4	19
5	It is useful to see the features of the messages sent and the type of the actor.	**4.37**	0.761	5	5	19
6	It would be useful to save the generated visualisation and play it without rerunning the program.	**4.53**	0.697	5	5	19
7	It would be useful to send a message to an actor from the web application.	**4.32**	0.582	4	4	19

Code frequencies are given in Table 7. From the code frequencies we can conclude that the students' general opinion of the system is positive. Students believe that AkkaVisual can assist them with learning the actor model. Some participants expressed their desire to use the system in class. One student pointed out that they "will use it [AkkaVisual] when taking the course", while another believes he/she would achieve a batter grade: "I think my results in the course would have been better with the use of this visualization".

The results obtained from this pilot study seem promising and motivate further development of the system. The main limitations of the study were the number of students and that they did not have an opportunity to use the visualization themselves.

X. VISUAL PROGRAMMING AND DATA SCIENCE

In this brief section we will discuss the application of visualizations for data science. The field has become increasingly popular in recent years, with deep learning models achieving record scores on benchmark problems. Python still seems to be one of the most popular languages for data science, with TensorFlow ("TensorFlow," n.d.), Pytorch ("PyTorch," n.d.) and Keras ("Home—Keras Documentation," n.d.) some of the most popular Deep Learning frameworks (Wongsuphasawat et al., 2018).

The goal in data science is to come up with a statistical (e.g. predictive) model that will accurately describe the given data. Experts often need to experiment with different models and parameters to achieve satisfactory results.

Some libraries, such as Keras, facilitate fast prototyping and experimentation by providing high-level APIs. That way, researchers do not need to implement the algorithms themselves. However, they still need to understand these complex models in order to optimize them.

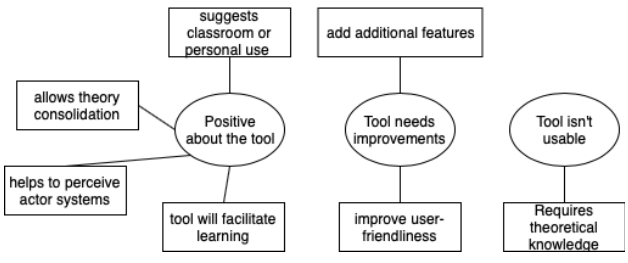

Fig. 5 A map of categories and codes derived by content analysis

Several APIs use the dataflow model to facilitate the development of deep learning models. TensorFlow Graph Visualizer is a tool that generates an interactive visualization of TensorFlow models. Visualizations are represented as dataflow graphs in which nodes are groups of operations. The system can assist developers with understanding their models and inspecting their structure. Users can also extend certain nodes to inspect their nested structure (Wongsuphasawat et al., 2018).

DeepVisual is a visual programming tool implemented as a PyCharm plugin. Instead of programming, developers can focus on the design of their deep learning models. The system provides different visual components to represent different types of layers, which can be connected to each other to construct deep neural networks. DeepVisual can generate the code from the visual graph structure, and vice-versa (Xie, Qi, Ma, & Zhao, 2019).

StatWire is an IDE for R which visualizes the transformation of data using a dataflow visual representation. One of the main goals of the system is to facilitate thinking in a modular way. The system provides two types of components: *statlets* as processing nodes, and *viewlets* for printing or plotting data. When the user creates a statlets an empty function in an R script is created. Code changes made by the user result in a live update of the visualization (Subramanian et al., 2018).

Orange3, is a dataflow visual language for designing machine learning and data mining applications. Its components, called widgets, support interactive data exploration. The system provides a widget for programming Python scripts and an API for adding new widgets. Orange3 can be used without any programming knowledge. The user simply drags the widgets and connects them (Demšar, Zupan, Leban, & Curk, 2004; "Orange3," n.d., p. 3).

TABLE 7 CODE FREQUENCIES OBTAINED WITH CONTENT ANALYSIS

Code	Frequency	Percentage
the tool will facilitate learning	5	45,45%
suggests classroom or personal use	3	27,27%
add additional features	2	18,18%
helps to perceive actor systems	2	18,18%
improve user-friendliness	2	18,18%
requires theoretical knowledge	1	9,09%
allows theory consolidation	1	9,09%

XI. Conclusion

In this paper we reported our findings from conducting a literature review on PV systems. We identified several new PV systems, some of which are research prototypes, while others could become long-term systems. Most of the evaluation papers that we reviewed reported that participants have a positive attitude towards PVs. Some papers also reported a positive effect on students' programming abilities.

We then discussed VPLs in general and provided an overview of several taxonomies. Our focus was on DFVPLs as a subcategory of VPLs. DFVPLs are often designed to provide non-programmers the ability to quickly develop specific types of applications. We presented PARVIS, a DFVPL that incorporates some features commonly found in PVs.

PARVIS was used with novice programmers and students learning functional programming. The system proved to be too complex for novice programmers. However, the average test score on assignments programmed in PARVIS were higher than those programmed in JavaScript. This result indicates that PARVIS could have a positive impact for learning functional programming and perhaps other more complex topics.

AkkaVisual was also described as a visualization system for actor programs. The system can also visualize the interactions between different programming entities such as objects and agents. Users simply need to provide the system with data that contains the required information in JSON, such as sender and receiver. The pilot study showed that students had a positive attitude towards AkkaVisual and would like to have used it for learning about the actor model.

For future work, we would like to extend the functionality of AkkaVisual. First, we would like to add some of the features that students scored in the questionnaire. Secondly, we would like to evaluate its usability in an experimental setting. Finally, we would like to develop additional components that would allow simpler visualization for OO and agent-based programs.

We would also like to inspect the usability of using existing simulation systems, such as NetLogo, as a visualization interface. Using NetLogo in this way could lead to a higher engagement level and more customization options for teachers. Teachers could prepare procedures that the student could use to view the visualization from a different perspective. Students could also use the NetLogo language to manipulate and further play with the visualizations.

References

Actors • Akka Documentation. (n.d.). Retrieved September 15, 2019, from https://doc.akka.io/docs/akka/current/actors.html

Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press.

Aglić Čuvić, M. (2018). *Introducing a Dataflow visual programming language for understanding program execution*. *2*(1), 35–40.

AguiaJ. (n.d.). Retrieved September 5, 2019, from http://www.aguiaj.org.pt/home

Al-Fedaghi, S., & Alrashed, A. (2014). Visualization of Execution of Programming Statements. *Proceedings of the 2014 11th International Conference on Information Technology: New Generations*, 363–370. https://doi.org/10.1109/ITNG.2014.74

AlFedaghi, S. S. (2019). Five Generic Processes for Behavior Description in Software Engineering. *ArXiv*, *abs/1907.11893*.

Al-Fedaghi, S., & Sultan, S. (2017). Flow Machine Diagrams for VHDL Code. *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, 162:1–162:6. https://doi.org/10.1145/3018896.3056779

Alireza Kavianpour. (2014). LabVIEW : A Teaching Tool for the Engineering Courses. *2014 ASEE Annual Conference & Exposition*, 24.842.1-24.842.11.

Al-Sakkaf, A., Omar, M., & Ahmad, M. (2019). A systematic literature review of student engagement in software visualization: A theoretical perspective. *Computer Science Education*, *29*(2–3), 283–309. https://doi.org/10.1080/08993408.2018.1564611

Annamaa, A. (2015). Introducing Thonny, a Python IDE for Learning Programming. *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, 117–121. https://doi.org/10.1145/2828959.2828969

Azadmanesh, M., & Hauswirth, M. (2017). Concept-Driven Generation of Intuitive Explanations of Program Execution for a Visual Tutor. *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, 64–73. https://doi.org/10.1109/VISSOFT.2017.22

Bácsi, S., & Mezei, G. (2019). Towards a Classification to Facilitate the Design of Domain-Specific Visual Languages. *Acta Cybernetica*, *24*(1), 5–16.

Banerjee, G., Murthy, S., & Iyer, S. (2013). Program visualization: Effect of viewing vs. Responding on student learning. *Proceedings of the 21st International Conference on Computers in Education, ICCE 2013*, 194–203. Retrieved from https://www.scopus.com/inward/record.uri?eid=2-s2.0-84896470578&partnerID=40&md5=6461d4f84556f81a980907a57f85dbe1

Banerjee, Gargi, Murthy, S., & Iyer, S. (2015). Effect of active learning using program visualization in technology-constrained college classrooms. *Research and Practice in Technology Enhanced Learning*, *10*(1), 15. https://doi.org/10.1186/s41039-015-0014-0

Ben-Ari, M. (1998). Constructivism in Computer Science Education. *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*, 257–261. https://doi.org/10.1145/273133.274308

Ben-Ari, M., Bednarik, R., Ben-Bassat Levy, R., Ebel, G., Moreno, A., Myller, N., & Sutinen, E. (2011). A decade of research and development on program animation: The Jeliot experience. *Journal of Visual Languages & Computing*, *22*(5), 375–384. https://doi.org/10.1016/j.jvlc.2011.04.004

Berney, S., & Bétrancourt, M. (2016). Does animation enhance learning? A meta-analysis. *Computers & Education*, *101*, 150–167. https://doi.org/10.1016/j.compedu.2016.06.005

Berry, M., & Kölling, M. (2016). Novis: A Notional Machine Implementation for Teaching Introductory Programming. *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, 54–59. https://doi.org/10.1109/LaTiCE.2016.5

Berry, Michael, & Kölling, M. (2013). The Design and Implementation of a Notional Machine for Teaching Introductory Programming. *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, 25–28. https://doi.org/10.1145/2532748.2532765

Berry, Michael, & Kölling, M. (2014). The State of Play: A Notional Machine for Learning Programming. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 21–26. https://doi.org/10.1145/2591708.2591721

Blockly. (n.d.). Retrieved August 12, 2019, from Blockly website: https://developers.google.com/blockly/

Bruce-Lockhart, M., Crescenzi, P., & Norvell, T. (2009). Integrating test generation functionality into the Teaching Machine environment. *Electronic Notes in Theoretical Computer Science*, *224*, 115–124. https://doi.org/10.1016/j.entcs.2008.12.055

Bruce-Lockhart, M. P., & Norvell, T. S. (2000). Lifting the hood of the computer: Program animation with the Teaching Machine. *2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era (Cat. No.00TH8492)*, *2*, 831–835 vol.2. https://doi.org/10.1109/CCECE.2000.849582

Bruce-Lockhart, M. P., & Norvell, T. S. (2007). Developing Mental Models of Computer Programming Interactively Via the Web. *2007 37th Annual Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports*, S3H-3-S3H-8. https://doi.org/10.1109/FIE.2007.4418051

Burnett, M. M., & Baker, M. J. (1993). *A Classification System for Visual Programming Languages*. Corvallis, OR, USA: Oregon State University.

Clang C Language Family Frontend for LLVM. (n.d.). Retrieved August 30, 2019, from https://clang.llvm.org/

CLIP. (n.d.). Retrieved September 1, 2019, from http://www.cs.tut.fi/~vip/clip/clip_english.html

Čolak, A., & Čuvić, M. A. (2019). An educational tool for visualising actor programs. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics*

(MIPRO), 605–610. https://doi.org/10.23919/MIPRO.2019.8756918

Cross, J., Hendrix, D., Barowski, L., & Umphress, D. (2014). Dynamic Program Visualizations: An Experience Report. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 609–614. https://doi.org/10.1145/2538862.2538958

Čuvić, M. A., Maras, J., & Mladenović, S. (2017). Extending the object-oriented notional machine notation with inheritance, polymorphism, and GUI events. *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 794–799. https://doi.org/10.23919/MIPRO.2017.7973530

Demšar, J., Zupan, B., Leban, G., & Curk, T. (2004). Orange: From Experimental Machine Learning to Interactive Data Mining. In J.-F. Boulicaut, F. Esposito, F. Giannotti, & D. Pedreschi (Eds.), *Knowledge Discovery in Databases: PKDD 2004* (pp. 537–539). Springer Berlin Heidelberg.

Dien, H. E., & Asnar, Y. D. W. (2018). OPT+Graph: Detection of Graph Data Structure on Program Visualization Tool to Support Learning. *2018 5th International Conference on Data and Software Engineering (ICoDSE)*, 1–6. https://doi.org/10.1109/ICODSE.2018.8705794

Du Boulay, B. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, *2*(1), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

Earwood, B., Jeong Yang, & Young Lee. (2016). Impact of static and dynamic visualization in improving object-oriented programming concepts. *2016 IEEE Frontiers in Education Conference (FIE)*, 1–5. https://doi.org/10.1109/FIE.2016.7757639

Egan, M. H., & McDonald, C. (2013). Runtime error checking for novice C programmers. *4th Annual International Conference on Computer ScienceEducation: Innovation and Technology (CSEIT 2013)*, 1–9.

Egan, M. H., & McDonald, C. (2014). Program Visualization and Explanation for Novice C Programmers. *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, 51–57. Retrieved from http://dl.acm.org/citation.cfm?id=2667490.2667496

Egan, M. H., & McDonald, C. (2015). Dynamic evaluation trees for novice C programmers. *Proceedings of the 17th Australasian Computing Education Conference (ACE 2015)*, *27*, 30.

Elvina, E., Karnalim, O., Ayub, M., & Wijanto, M. C. (2018). Combining program visualization with programming workspace to assist students for completing programming laboratory task. *JOTSE: Journal of Technology and Science Education*, *8*(4), 268–280.

Eranki, K. L. N., & Moudgalya, K. M. (2013). An Integrated Approach to Build Programming Competencies through Spoken Tutorial Workshops. *2013 IEEE Fifth International Conference on Technology for Education (T4e 2013)*, 28–31. https://doi.org/10.1109/T4E.2013.15

Erwig, M., Smeltzer, K., & Wang, X. (2017). What is a Visual Language? *J. Vis. Lang. Comput.*, *38*I, 9–17. https://doi.org/10.1016/j.jvlc.2016.10.005

Gestwicki, P., & Jayaraman, B. (2005). Methodology and Architecture of JIVE. *Proceedings of the 2005 ACM Symposium on Software Visualization*, 95–104. https://doi.org/10.1145/1056018.1056032

Gestwicki, P. V. (2004). Interactive Visualization of Object-oriented Programs. *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 48–49. https://doi.org/10.1145/1028664.1028691

Gordon, M., & Guo, P. J. (2015). Codepourri: Creating visual coding tutorials using a volunteer crowd of learners. *2015 IEEE Symposium on Visual*

Languages and Human-Centric Computing (VL/HCC), 13–21. https://doi.org/10.1109/VLHCC.2015.7357193

Guo, P. J., White, J., & Zanelatto, R. (2015). Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 79–87. https://doi.org/10.1109/VLHCC.2015.7357201

Guo, Philip J. (2013). Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 579–584. https://doi.org/10.1145/2445196.2445368

Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Terasvirta, T., & Vanninen, P. (1997). Animation of user algorithms on the Web. *Proceedings. 1997 IEEE Symposium on Visual Languages (Cat. No.97TB100180)*, 356–363. https://doi.org/10.1109/VL.1997.626605

Hendrix, T. D., Cross, J. H., II, & Barowski, L. A. (2004). An Extensible Framework for Providing Dynamic Data Structure Visualizations in a Lightweight IDE. *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, 387–391. https://doi.org/10.1145/971300.971433

Hewitt, C., Bishop, P., & Steiger, R. (1973). A Universal Modular ACTOR Formalism for Artificial Intelligence. *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, 235–245. Retrieved from http://dl.acm.org/citation.cfm?id=1624775.1624804

Hidalgo-Céspedes, J., Marín-Raventós, G., & Lara-Villagrán, V. (2016). Learning principles in program visualizations: A systematic literature review. *2016 IEEE Frontiers in Education Conference (FIE)*, 1–9. https://doi.org/10.1109/FIE.2016.7757692

Hils, D. D. (1992). Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, *3*(1), 69–101. https://doi.org/10.1016/1045-926X(92)90034-J

Home—Keras Documentation. (n.d.). Retrieved September 18, 2019, from https://keras.io/

Hong, H., Oh, H., & Ha, S. (2017). Hierarchical Dataflow Modeling of Iterative Applications. *Proceedings of the 54th Annual Design Automation Conference 2017*, 39:1–39:6. https://doi.org/10.1145/3061639.3062260

Hosseini, R., Sirkiä, T., Guerra, J., Brusilovsky, P., & Malmi, L. (2016). Animated Examples As Practice Content in a Java Programming Course. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 540–545. https://doi.org/10.1145/2839509.2844639

Hundhausen, C. D., Douglas, S. A., & Stasko, J. T. (2002). A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing*, *13*(3), 259–290. https://doi.org/10.1006/jvlc.2002.0237

Ihara, D., KOGURE, S., Noguchi, Y., Yamashita, K., Konishi, T., & Itoh, Y. (2017). Algorithm Learning by Comparing Visualized Behavior of Programs. *Proceedings of the 25th International Conference on Computers in Education*.

Ishizue, R., Sakamoto, K., Washizaki, H., & Fukazawa, Y. (2018). PVC: Visualizing C Programs on Web Browsers for Novices. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 245–250. https://doi.org/10.1145/3159450.3159566

Isohanni, E., & Knobelsdorf, M. (2013). *Students' Engagement with the Visualization Tool VIP in Light of Activity Theory*. Tampere University of Technology. Department of Pervasive Computing.

Jayaraman, S., Jayaraman, B., & Lessa, D. (2017). Compact Visualization of Java Program Execution. *Softw. Pract. Exper.*, *47*(2), 163–191. https://doi.org/10.1002/spe.2411

Jeong Yang, Young Lee, & Hicks, D. (2016). Synchronized static and dynamic visualization in a web-based

programming environment. *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 1–4. https://doi.org/10.1109/ICPC.2016.7503733

JGRASP Home Page. (n.d.). Retrieved August 27, 2019, from https://www.jgrasp.org/index.html

Johnston, W. M., Hanna, J. R. P., & Millar, R. J. (2004). Advances in Dataflow Programming Languages. *ACM Comput. Surv.*, *36*(1), 1–34. https://doi.org/10.1145/1013208.1013209

Kaila, E., Rajala, T., Laakso, M.-J., & Salakoski, T. (2008). Automatic Assessment of Program Visualization Exercises. *Proceedings of the 8th International Conference on Computing Education Research*, 101–104. https://doi.org/10.1145/1595356.1595376

Kang, H., & Guo, P. J. (2017). Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 737–745. https://doi.org/10.1145/3126594.3126632

Karnalim, O., & Ayub, M. (2017a). The Effectiveness of a Program Visualization Tool on Introductory Programming: A Case Study with PythonTutor. *CommIT (Communication and Information Technology) Journal*, *11*(2), 67–76.

Karnalim, O., & Ayub, M. (2017b). The Use of Python Tutor on Programming Laboratory Session: Student Perspectives. *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control*, *2*(4), 327–336.

Karnalim, O., & Ayub, M. (2018). A Quasi-Experimental Design to Evaluate the Use of PythonTutor on Programming Laboratory Session. *International Journal of Online Engineering (IJOE)*, *14*(02), 155. https://doi.org/10.3991/ijoe.v14i02.8067

Kumalija, E. J., Fatih, Y., & Sun, Y. (2019). STUDENTS' PERCEPTION TOWARDS PROGRAM VISUALIZATION ON SMARTPHONE - CASE OF SUNLAB INITIAL INVESTIGATION. *15th International Conference on Mobile Learning 2019*, 42–48. https://doi.org/10.33965/ml2019_201903L006

Kumalija, E., Yi, S., & Fatih, Y. (2018). Dynamic Program Visualization on Android Smartphones for Novice Java Programmers. *International Association for Development of the Information Society*.

Laakso, M.-J., Kaila, E., & Rajala, T. (2018). ViLLE — Collaborative Education Tool: Designing and Utilizing an Exercise-based Learning Environment. *Education and Information Technologies*, *23*(4), 1655–1676. https://doi.org/10.1007/s10639-017-9659-1

Lahtinen, E., & Ahoniemi, T. (2005). Visualizations to support programming on different levels of cognitive development. *Proceedings of the Koli Calling 2005 Conference on Computer Science Education, November 2005, Koli, Finland*, 87–94.

Lahtinen, S.-, Sutinen, E., Tarhio, J., & Tuovinen, A.-. (1997). Object-oriented visualization of program logic. *Proceedings of TOOLS USA 97. International Conference on Technology of Object Oriented Systems and Languages*, 76–88. https://doi.org/10.1109/TOOLS.1997.654702

Lee, E. A., & Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE*, *83*(5), 773–801. https://doi.org/10.1109/5.381846

LEGO® Education WeDo 2.0 Core Set. (n.d.). Retrieved September 9, 2019, from https://education.lego.com/en-us/products/lego-education-wedo-2-0-core-set-/45300

Levy, R. B.-B., Ben-Ari, M., & Uronen, P. A. (2003). The Jeliot 2000 program animation system. *Computers & Education*, *40*(1), 1–15. https://doi.org/10.1016/S0360-1315(02)00076-3

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin*, *38*(3), 118–122.

Luxton-Reilly, A. (2016). Learning to Program is Easy. *Proceedings of the 2016 ACM Conference on*

Innovation and Technology in Computer Science Education, 284–289. https://doi.org/10.1145/2899415.2899432

Luxton-Reilly, A., Simon, Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., … Szabo, C. (2018). Introductory Programming: A Systematic Literature Review. *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, 55–106. https://doi.org/10.1145/3293881.3295779

Ma, L., Ferguson, J. D., Roper, M., Ross, I., & Wood, M. (2008). Using Cognitive Conflict and Visualisation to Improve Mental Models Held by Novice Programmers. *SIGCSE Bull.*, *40*(1), 342–346. https://doi.org/10.1145/1352322.1352253

Ma, L., Ferguson, J., Roper, M., Ross, I., & Wood, M. (2009). Improving the Mental Models Held by Novice Programmers Using Cognitive Conflict and Jeliot Visualisations. *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, 166–170. https://doi.org/10.1145/1562877.1562931

Maletic, J. I., Marcus, A., & Collard, M. L. (2002). A task oriented view of software visualization. *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis*, 32–40. https://doi.org/10.1109/VISSOF.2002.1019792

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch Programming Language and Environment. *Trans. Comput. Educ.*, *10*(4), 16:1–16:15. https://doi.org/10.1145/1868358.1868363

Mason, D., & Dave, K. (2017). Block-based versus flow-based programming for naive programmers. *2017 IEEE Blocks and Beyond Workshop (B B)*, 25–28. https://doi.org/10.1109/BLOCKS.2017.8120405

Moons, J., & De Backer, C. (2013). The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Computers & Education*, *60*(1), 368–384. https://doi.org/10.1016/j.compedu.2012.08.009

Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004). Visualizing Programs with Jeliot 3. *Proceedings of the Working Conference on Advanced Visual Interfaces*, 373–376. https://doi.org/10.1145/989863.989928

Moreno, A., Sutinen, E., Bednarik, R., & Myller, N. (2007). Conflictive Animations As Engaging Learning Tools. *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88*, 203–206. Retrieved from http://dl.acm.org/citation.cfm?id=2449323.2449352

Moreno, A., Sutinen, E., & Joy, M. (2014). Defining and Evaluating Conflictive Animations for Programming Education: The Case of Jeliot ConAn. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 629–634. https://doi.org/10.1145/2538862.2538888

Moreno, A., Sutinen, E., & Sedano, C. I. (2013). A game concept using conflictive animations for learning programming. *2013 IEEE International Games Innovation Conference (IGIC)*, 175–178. IEEE.

Mosconi, M., & Porta, M. (2000). Iteration constructs in data-flow visual programming languages. *Computer Languages*, *26*(2), 67–104. https://doi.org/10.1016/S0096-0551(01)00009-1

Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, *1*(1), 97–123. https://doi.org/10.1016/S1045-926X(05)80036-9

Myller, N., Bednarik, R., Sutinen, E., & Ben-Ari, M. (2009). Extending the Engagement Taxonomy: Software Visualization and Collaborative Learning. *Trans. Comput. Educ.*, *9*(1), 7:1–7:27. https://doi.org/10.1145/1513593.1513600

Nagae, A., & Kagawa, K. (2014). DEVELOPMENT OF A VISUAL DEBUGGER FOR C IMPLEMENTED IN JAVASCRIPT. *Theory and Practice of Computation: Proceedings of Workshop on*

Computation: Theory and Practice WCTP2013, 208–217. https://doi.org/DOI: 10.1142/9789814612883_0015

Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., … Velázquez-Iturbide, J. Á. (2002). Exploring the Role of Visualization and Engagement in Computer Science Education. *SIGCSE Bull.*, *35*(2), 131–152. https://doi.org/10.1145/782941.782998

Nascimento, M. D. do, Oliveira, F. C. d M. B., Alves, S. S. A., Freitas, A. T. de, Gomes, L. A. C. B., & Matos, A. S. de. (2017). A comparative study of deaf and non-deaf students' performance when using a Visual Java Debugger. *2017 IEEE Frontiers in Education Conference (FIE)*, 1–8. https://doi.org/10.1109/FIE.2017.8190514

Nathasya, R. A., Karnalim, O., & Ayub, M. (2019). Integrating program and algorithm visualisation for learning data structure implementation. *Egyptian Informatics Journal*. https://doi.org/10.1016/j.eij.2019.05.001

Noone, M., & Mooney, A. (2018). Visual and textual programming languages: A systematic review of the literature. *Journal of Computers in Education*, *5*(2), 149–174. https://doi.org/10.1007/s40692-018-0101-5

Orange3. (n.d.). Retrieved April 30, 2018, from https://orange.biolab.si

Pareja-Flores, C., Urquiza-Fuentes, J., & Velázquez-Iturbide, J. Á. (2007). WinHIPE: An IDE for Functional Programming Based on Rewriting and Visualization. *SIGPLAN Not.*, *42*(3), 14–23. https://doi.org/10.1145/1273039.1273042

Pawelczak, D., & Baumann, A. (2014). Virtual-C - a programming environment for teaching C in undergraduate programming courses. *2014 IEEE Global Engineering Education Conference (EDUCON)*, 1142–1148. https://doi.org/10.1109/EDUCON.2014.7096836

Pawelczak, Dieter. (2016). Benefits of a Testing Framework in Undergraduate C Programming Courses. *2nd International Conference on Higher Education Advances,HEAd'16, 21-23 June 2016, València, Spain*, *228*, 215–221. https://doi.org/10.1016/j.sbspro.2016.07.032

Pawelczak, Dieter, Baumann, A., & Schmudde, D. (2015). A new Testing Framework for C-Programming Exercises and Online-Assessments. *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, 279. The Steering Committee of The World Congress in Computer Science, Computer ….

Pears, A., & Rogalli, M. (2011). mJeliot: A Tool for Enhanced Interactivity in Programming Instruction. *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, 16–22. https://doi.org/10.1145/2094131.2094135

Price, B. A., Baecker, R. M., & Small, I. S. (1993). A Principled Taxonomy of Software Visualization. *Journal of Visual Languages & Computing*, *4*(3), 211–266. https://doi.org/10.1006/jvlc.1993.1015

PyCharm: The Python IDE for Professional Developers by JetBrains. (n.d.). Retrieved August 28, 2019, from JetBrains website: https://www.jetbrains.com/pycharm/

PyTorch. (n.d.). Retrieved September 18, 2019, from https://www.pytorch.org

Real-time ASP.NET with SignalR | .NET. (n.d.). Retrieved September 17, 2019, from https://dotnet.microsoft.com/apps/aspnet/signalr

Roman, G.-, & Cox, K. C. (1993). A taxonomy of program visualization systems. *Computer*, *26*(12), 11–24. https://doi.org/10.1109/2.247643

S. Xinogalos. (2013). Using flowchart-based programming environments for simplifying programming and software engineering processes. *2013 IEEE Global Engineering Education Conference (EDUCON)*, 1313–1322. https://doi.org/10.1109/EduCon.2013.6530276

Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs.

Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments, 37–39. https://doi.org/10.1109/HCC.2002.1046340

Sajaniemi, Jorma, Byckling, P., & Gerdt, P. (2006). Metaphor-based Animation of OO Programs. *Proceedings of the 2006 ACM Symposium on Software Visualization*, 173–174. https://doi.org/10.1145/1148493.1148530

Sajaniemi, Jorma, Byckling, P., & Gerdt, P. (2007). Animation Metaphors for Object-Oriented Concepts. *Electronic Notes in Theoretical Computer Science*, *178*, 15–22. https://doi.org/10.1016/j.entcs.2007.01.037

Sajaniemi, Jorma, & Kuittinen, M. (2003). Program Animation Based on the Roles of Variables. *Proceedings of the 2003 ACM Symposium on Software Visualization*, 7–ff. https://doi.org/10.1145/774833.774835

Santos, A. L. (2018). Enhancing Visualizations in Pedagogical Debuggers by Leveraging on Code Analysis. *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, 11:1–11:9. https://doi.org/10.1145/3279720.3279732

Santos, A. L., & Sousa, H. S. (2017). PandionJ: A Pedagogical Debugger Featuring Illustrations of Variable Tracing and Look-ahead. *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, 195–196. https://doi.org/10.1145/3141880.3141911

Schumacher, R. M., & Czerwinski, M. P. (1992). Mental Models and the Acquisition of Expert Knowledge. In R. R. Hoffman (Ed.), *The Psychology of Expertise* (pp. 61–79).

SeeC. (n.d.). Retrieved August 30, 2019, from https://seec-team.github.io/seec/seec-view.html

Shi, N., Min, Z., & Zhang, P. (2017). Effects of visualizing roles of variables with animation and IDE in novice program construction. *Telematics and Informatics*, *34*(5), 743–754. https://doi.org/10.1016/j.tele.2017.02.005

Shin, W.-C. (2018). A Study on the Effects of Visualization Tools on Debugging Program and Extending Functionality. *International Journal of Advanced Science and Technology*, *115*, 149–160. https://doi.org/10.14257/ijast.2018.115.14

Silva, L. C., Oliveira, F. C. d M. B., Oliveira, A. C. d, & Freitas, A. T. d. (2014). Introducing the JLoad: A Java Learning Object to Assist the Deaf. *2014 IEEE 14th International Conference on Advanced Learning Technologies*, 579–583. https://doi.org/10.1109/ICALT.2014.169

Sirkiä, T. (2016). Jsvee & Kelmu: Creating and Tailoring Program Animations for Computing Education. *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, 36–45. https://doi.org/10.1109/VISSOFT.2016.24

Sirkiä, T., & Sorva, J. (2015). Tailoring Animations of Example Programs. *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, 147–151. https://doi.org/10.1145/2828959.2828965

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), 1–31. https://doi.org/10.1145/2483710.2483713

Sorva, J., Karavirta, V., & Malmi, L. (2013). A Review of Generic Program Visualization Systems for Introductory Programming Education. *Trans. Comput. Educ.*, *13*(4), 15:1–15:64. https://doi.org/10.1145/2490822

Sorva, J., Lönnberg, J., & Malmi, L. (2013). Students' ways of experiencing visual program simulation. *Computer Science Education*, *23*(3), 207–238. https://doi.org/10.1080/08993408.2013.807962

Sorva, J., & Sirkia, T. (2010). UUhistle: A software tool for visual program simulation. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research Koli Calling 10*, 49–54. https://doi.org/10.1145/1930464.1930471

Sousa, T. B. (2012). Dataflow programming concept, languages and applications. *Doctoral Symposium on Informatics Engineering*, *130*.

Stasko, J. T., & Patterson, C. (1992). Understanding and characterizing software visualization systems. *Proceedings IEEE Workshop on Visual Languages*, 3–10. https://doi.org/10.1109/WVL.1992.275790

Stefik, A., & Hanenberg, S. (2014). The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 283–299. https://doi.org/10.1145/2661136.2661156

Subramanian, K., Maas, J., Ellers, M., Wacharamanotham, C., Voelker, S., & Borchers, J. (2018). StatWire: Visual Flow-based Statistical Programming. *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, LBW104:1–LBW104:6. https://doi.org/10.1145/3170427.3188528

Sulistiani, L., & Karnalim, O. (2018). An Embedding Technique for Language-Independent Lecturer-Oriented Program Visualization. *EMITTER International Journal of Engineering Technology*, *6*(1), 92–104. https://doi.org/10.24003/emitter.v6i1.234

Suzuki, R., Soares, G., Head, A., Glassman, E., Reis, R., Mongiovi, M., … Hartmann, B. (2017). TraceDiff: Debugging unexpected code behavior using trace divergences. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 107–115. https://doi.org/10.1109/VLHCC.2017.8103457

Tang, T., Rixner, S., & Warren, J. (2014). An Environment for Learning Interactive Programming. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 671–676. https://doi.org/10.1145/2538862.2538908

Tanimoto, S. L. (1990). VIVA: A Visual Language for Image Processing. *J. Vis. Lang. Comput.*, *1*(2), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6

Tekdal, M. (2013). The Effect of an Example-Based Dynamic Program Visualization Environment on Students' Programming Skills. *Journal of Educational Technology & Society*, *16*(3), 400–410.

TensorFlow. (n.d.). Retrieved September 18, 2019, from TensorFlow website: https://www.tensorflow.org/

Tezuka, D., Kogure, S., Noguchi, Y., Yamashita, K., Konishi, T., & Itoh, Y. (2016). *GUI based environment to support writing and debugging rules for a program visualization tool*. 303–305. Retrieved from https://www2.scopus.com/inward/record.uri?eid=2-s2.0-85019007751&partnerID=40&md5=ea514a8f4ba9ddf5c7b0c91da9105bf9

Thayer, K., Guo, P. J., & Reinecke, K. (2018). The Impact of Culture on Learner Behavior in Visual Debuggers. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.

Urquiza-Fuentes, J., & Velázquez-Iturbide, J. Á. (2012). Comparing the effectiveness of different educational uses of program animations. *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, 174–179. ACM.

Urquiza-Fuentes, J., & Velázquez-Iturbide, J. Á. (2013). Toward the effective use of educational program animations: The roles of student's engagement and topic complexity. *Computers & Education*, *67*, 178–192. https://doi.org/10.1016/j.compedu.2013.02.013

Wongsuphasawat, K., Smilkov, D., Wexler, J., Wilson, J., Mané, D., Fritz, D., … Wattenberg, M. (2018). Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow. *IEEE Transactions on Visualization and Computer Graphics*, *24*(1), 1–12. https://doi.org/10.1109/TVCG.2017.2744878

Xie, C., Qi, H., Ma, L., & Zhao, J. (2019). DeepVisual: A Visual Programming Tool for Deep Learning Systems. *Proceedings of the 27th International Conference on Program Comprehension*, 130–134. https://doi.org/10.1109/ICPC.2019.00028

Xue, G., Yang, Q., & Xing, J. (2017). A survey of graphical programming language and its applications in intelligent buildings. *2017 Chinese Automation Congress (CAC)*, 6822–6828. https://doi.org/10.1109/CAC.2017.8244006

Yamamoto, R., Anzai, Y., Kogure, S., Noguchi, Y., Yamashita, K., Konishi, T., & Itoh, Y. (2017). *Learning Environment for Recursive Functions by Visualization of Execution Process*.

Yamashita, K., Fujioka, R., Kogure, S., Noguchi, Y., Konishi, T., & Itoh, Y. (2015). Educational Practice of Algorithm using Learning Support System with Visualization of Program Behavior. *Proceedings of the 23rd International Conference on Computers in Education*, 632–640.

Yamashita, K., Fujioka, R., Kogure, S., Noguchi, Y., Konishi, T., & Itoh, Y. (2016). Practices of algorithm education based on discovery learning using a program visualization system. *Research and Practice in Technology Enhanced Learning*, *11*(1), 15. https://doi.org/10.1186/s41039-016-0041-5

Yamashita, K., Fujioka, R., Kogure, S., Noguchi, Y., Konishi, T., & Itoh, Y. (2017). Classroom practice for understanding pointers using learning support system for visualizing memory image and target domain world. *Research and Practice in Technology Enhanced Learning*, *12*(1), 17. https://doi.org/10.1186/s41039-017-0058-4

Yamashita, K., Tezuka, D., Kogure, S., Noguchi, Y., Konishi, T., & Itoh, Y. (2018). A Learning Support System for Visualizing Behaviors of Students' Programs Based on Teachers' Intents of Instruction. *Proceedings of the 26th International Conference on Computers in Education*, 761–766.

YAN, Y., HARA, K., KAZUMA, T., HISADA, Y., & HE, A. (2018). PROVIT-CI: A Classroom-Oriented Educational Program Visualization Tool. *IEICE Transactions on Information and Systems*, *E101.D*(2), 447–454. https://doi.org/10.1587/transinf.2017EDK0002

Yan, Y., Nakano, H., Hara, K., Kazuma, T., & He, A. (2016). A Web Service for C Programming Learning and Teaching. *2016 10th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*, 414–419. https://doi.org/10.1109/CISIS.2016.70

Yan, Yu, Hiroto, N., Kohei, H., Shota, S., & He, A. (2014). A C Programming Learning Support System and Its Subjective Assessment. *Proceedings of the 2014 IEEE International Conference on Computer and Information Technology*, 561–566. https://doi.org/10.1109/CIT.2014.23

Yang, J., Lee, Y., & Chang, K. H. (2017). Initial Evaluation of JaguarCode: A Web-Based Object-Oriented Programming Environment with Static and Dynamic Visualization. *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE T)*, 152–161. https://doi.org/10.1109/CSEET.2017.32

Yang, J., Lee, Y., Gandhi, D., & Valli, S. G. (2017). Synchronized UML diagrams for object-oriented program comprehension. *2017 12th International Conference on Computer Science and Education (ICCSE)*, 12–17. https://doi.org/10.1109/ICCSE.2017.8085455

Yang, J., Lee, Y., Hicks, D., & Chang, K. H. (2015). Enhancing object-oriented programming education using static and dynamic visualization. *2015 IEEE Frontiers in Education Conference (FIE)*, 1–5. https://doi.org/10.1109/FIE.2015.7344152

Yang, Jeong-sug. (2016). *JavelinaCode: A Web-Based Object-Oriented Programming Environment with Static and Dynamic Visualization* (PhD Dissertation).

Zhang, K. (2007). *Visual Languages and Applications*. Retrieved from https://www.springer.com/gp/book/9780387298139

Ziarek, L., Jayaraman, B., Lessa, D., & Swaminathan, J. (2016). Runtime Visualization and Verification in JIVE. In Y. Falcone & C. Sánchez (Eds.), *Runtime Verification* (pp. 493–497). Cham: Springer International Publishing.

APPENDIX

The table in this appendix contains a list of the PV related papers that we identified through literature search.

TABLE 8 PAPERS IDENTIFIED THROUGH LITERATURE REVIEW

#	Title	Authors	Publication year	PV
1	Novis A notional machine implementation for teaching introductory programming	Berry, M., & Kölling, M.	2016	BlueJ Novis
2	The state of play: A notional machine for learning programming	Berry, M., Kölling, M.	2014	BLueJ Novis
3	The design and implementation of a notional machine for teaching introductory programming	Berry, M., Kölling, M.	2013	BlueJ Novis
4	An environment for learning interactive programming	Tang, T., Rixner, S., & Warren, J.	2014	CodeSkulptor
5	The effect of an example-based dynamic program visualization environment on students' programming skills	Tekdal, M.	2013	EDPVE
6	Visualization of execution of programming statements	Al-Fedaghi, S., Alrashed, A.	2014	FM Visualization/TM Visualization
7	A comparative study of deaf and non-deaf students' performance when using a visual Java debugger	Do Nascimento, M.D., Oliveira, F.C.D.M.B., Alves, S.S.A., De Freitas, A.T., Gomes, L.A.C.B., De Matos, A.S.	2017	JAD & Jload
8	Synchronized Static and Dynamic Visualization in a Web-Based Programming Environment	Jeong Yang, Young Lee, & Hicks, D.	2016	JaguarCode
9	Initial Evaluation of JaguarCode: A Web-Based Object-Oriented Programming Environment with Static and Dynamic Visualization	Yang, J., Lee, Y., & Chang, K. H.	2017	JavelinaCode/JaguarCode
10	Synchronized UML diagrams for object-oriented program comprehension	J. Yang; Y. Lee; D. Gandhi; S. G. Valli	2017	JavelinaCode/JaguarCode
11	Impact of Static and Dynamic Visualization in Improving OOP Concepts	Earwood, B., Jeong Yang, & Young Lee.	2016	JavelinaCode/JaguarCode
12	JavelinaCode- A Web-Based Object-Oriented Programming Environment with Static and Dynamic Visualization	Yang, Jeong-sug.	2016	JavelinaCode/JaguarCode
13	Enhancing Object-Oriented Programming Education using Static and Dynamic Visualization	Yang, J., Lee, Y., Hicks, D., & Chang, K. H.	2015	JavelinaCode/JaguarCode
14	Defining and evaluating conflictive animations for programming education: The case of Jeliot ConAn	Moreno, A., Sutinen, E., Joy, M.	2014	Jeliot ConAn
15	A game concept using conflictive animations for learning programming	Moreno, Andres; Sutinen, Erkki; Sedano, Carolina Islas	2013	Jeliot ConAn
16	Dynamic program visualizations - An experience report	Cross II, J.H., Hendrix, T.D., Barowski, L.A., Umphress, D.A.	2014	jGrasp
17	Compact visualization of Java program execution	Jayaraman, S., Jayaraman, B., & Lessa, D.	2017	JIVE
18	Runtime Visualization and Verification in JIVE	Ziarek, L., Jayaraman, B., Lessa, D., & Swaminathan, J.	2016	JIVE
19	An Embedding Technique for Language-Independent Lecturer-Oriented Program Visualization Tool	Sulistiani, Lisan; Karnalim, Oscar	2018	LISN
20	A study on the effects of visualization tools on debugging program and extending functionality	Shin, W.-C.	2018	ObjectVisualizer
21	Omnicode A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations	Kang, H., & Guo, P. J.	2017	Omnicode
22	A quasi-experimental design to evaluate the use of pythontutor on programming laboratory session	Karnalim, O., Ayub, M.	2018	OPT
23	The impact of culture on learner behavior in visual debuggers	Thayer, K., Guo, P.J., Reinecke, K.	2018	OPT
24	The Effectiveness of a Program Visualization Tool on Introductory Programming - A Case Study with PythonTutor	Karnalim, O., & Ayub, M.	2017	OPT
25	The Use of PythonTutor on Programming Laboratory Session- Student Perspectives	Karnalim, O., & Ayub, M.	2017	OPT
26	Online python tutor: Embeddable web-based program visualization for cs education	Guo, P.J.	2013	OPT

27	Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning	Guo, P.J., White, J., Zanelatto, R.	2015	OPT (addon)
28	Codepourri: Creating visual coding tutorials using a volunteer crowd of learners	Gordon, M., Guo, P.J.	2015	OPT (addon)
29	Concept-Driven Generation of Intuitive Explanations of Program Execution for a Visual Tutor	Azadmanesh, M., Hauswirth, M.	2017	OPT (codeexplanations)
30	OPT+ Graph: Detection of Graph Data Structure on Program Visualization Tool to Support Learning	Dien, Habibie Ed; Asnar, Yudistira Dwi Wardhana	2018	OPT (graph data structure)
31	TraceDiff: Debugging unexpected code behavior using trace divergences	R. Suzuki; G. Soares; A. Head; E. Glassman; R. Reis; M. Mongiovi; L. D'Antoni; B. Hartmann	2017	OPT (tracediff)
32	Enhancing Visualizations in Pedagogical Debuggers by Leveraging on Code Analysis	Santos, A. L.	2018	PandionJ
33	PandionJ- a pedagogical debugger featuring illustrations of variable tracing and look-ahead	Santos, A. L., & Sousa, H. S.	2017	PandionJ
34	Integrating program and algorithm visualisation for learning data structure implementation	Nathasya, R.A., Karnalim, O., Ayub, M.	2019	PITON
35	Combining program visualization with programming workspace to assist students for completing programming laboratory task	Elvina, E., Karnalim, O., Ayub, M., Wijanto, M.C.	2018	PITON
36	Effects of Visualizing Roles of Variables with Animation and IDE in Novice Program Construction	Shi, N., Min, Z., & Zhang, P.	2017	PlanAni
37	PROVIT-CI: A Classroom-Oriented Educational Program Visualization Tool	Yan, Yu; Hara, Kohei; Kazuma, Takenobu; Hisada, Yasuhiro; He, Aiguo	2018	PROVIT
38	A Web Service for C Programming Learning and Teaching	Yan, Y., Nakano, H., Hara, K., Kazuma, T., & He, A.	2016	PROVIT
39	A C programming learning support system and its subjective assessment	Yan, Y., Hiroto, N., Kohei, H., Shota, S., He, A.	2014	PROVIT
40	PVC: Visualizing C programs on web browsers for novices	Ishizue, R., Washizaki, H., Sakamoto, K., Fukazawa, Y.	2018	PVC
41	A systematic literature review of student engagement in software visualization: a theoretical perspective	Al-Sakkaf, Abdullah; Omar, Mazni; Ahmad, Mazida	2019	Review
42	Learning principles in program visualizations: A systematic literature review	Hidalgo-Céspedes, J., Marín-Raventós, G., Lara-Villagrán, V.	2016	Review
43	A review of generic program visualization systems for introductory programming education	Sorva, J., Karavirta, V., Malmi, L.	2013	Review
44	Program Visualization and Explanation for Novice C Programmers	Matthew Heinsen Egan and Chris McDonald	2014	SeeC
45	Runtime Error Checking for Novice C Programmers	Egan, M. H., & McDonald, C.	2013	SeeC
46	Students' perception towards program visualization on smartphone - case of sunlab initial investigation	Kumalija, E.J., Fatih, Y., Sun, Y.	2019	SunLab
47	Dynamic program visualization on android smartphones for novice Java programmers	Kumalija, E., Yi, S., Fatih, Y.	2018	SunLab
48	A Learning Support System for Visualizing Behaviors of Students' Programs Based on Teachers' Intents of Instruction	Yamashita, Koichi; Tezuka, Daiki; Kogure, Satoru; Noguchi, Yasuhiro; Konishi, Tatsuhiro; Itoh, Yukihiro	2018	TEDViT
49	Algorithm Learning by Comparing Visualized Behavior of Programs	Ihara, Daiki; Kogure, Satoru; Noguchi, Yasuhiro; Yamashita, Koichi; Konishi, Tatsuhiro; Itoh, Yukihiro	2017	TEDViT
50	Classroom practice for understanding pointers using learning support system for visualizing memory image and target domain world	Yamashita, K., Fujioka, R., Kogure, S., Noguchi, Y., Konishi, T., Itoh, Y.	2017	TEDViT
51	Learning Environment for Recursive Functions by Visualization of Execution Process	Yamamoto, Raiya; Anzai, Yasuhiro; Kogure, Satoru; Noguchi, Yasuhiro; Yamashita, Koichi; Konishi, Tatsuhiro; Itoh, Yukihiro	2017	TEDViT
52	GUI based environment to support writing and debugging rules for a program visualization tool	Tezuka, D., Kogure, S., Noguchi, Y., Yamashita, K., Konishi, T., Itoh, Y.	2016	TEDViT
53	Practices of algorithm education based on discovery learning using a program visualization system	Yamashita, K., Fujioka, R., Kogure, S., Noguchi, Y., Konishi, T., Itoh, Y.	2016	TEDViT
54	Educational Practice of Algorithm using Learning Support System with Visualization of Program Behavior	Yamashita, K., Fujioka, R., Kogure, S., Noguchi, Y., Konishi, T., & Itoh, Y.	2015	TEDViT
55	Introducing Thonny, a Python IDE for Learning Programming	Aivar Annamaa	2015	Thonny
56	Students' ways of experiencing visual program simulation	Sorva, J., Lönnberg, J., Malmi, L.	2013	UUhistle

57	Ville--collaborative-education-tool	Laakso, M.-J., Kaila, E., & Rajala, T.	2018	Ville
58	Students' engagement with the visualization tool VIP in light of activity theory	Isohanni, E., & Knobelsdorf, M.	2013	VIP
59	Benefits of a testing framework in undergraduate C programming courses	Pawelczak, Dieter	2016	Virtual-C
60	A new Testing Framework for C-Programming Exercises and Online-Assessments	Pawelczak, Dieter, Baumann, A., & Schmudde, D.	2015	Virtual-C
61	Virtual-C - a programming environment for teaching C in undergraduate programming courses	Pawelczak, Dieter; Baumann, Andrea	2014	Virtual-C
62	Toward the effective use of educational program animations: The roles of student's engagement and topic complexity	Urquiza-Fuentes, J., Velázquez-Iturbide, J.Á.	2013	WinHIPE
63	DEVELOPMENT OF A VISUAL DEBUGGER FOR C IMPLEMENTED IN JAVASCRIPT	Nagae, Akihiko; Kagawa, Koji	2015	(Nagae, Koji)
64	An integrated approach to build programming competencies through spoken tutorial workshops	Eranki, K.L.N., Moudgalya, K.M.	2013	