

# Program State Visualizer with User-Defined Representation Conversion (WIP)

Rifqi Adlan Apriyadi  
apriyadi.r.aa@m.titech.ac.jp  
Tokyo Institute of Technology  
Tokyo, Japan

Hidehiko Masuhara  
masuhara@acm.org  
Tokyo Institute of Technology  
Tokyo, Japan

Youyou Cong  
cong@c.titech.ac.jp  
Tokyo Institute of Technology  
Tokyo, Japan

## ABSTRACT

Conventional non-visual tree-based debuggers possess comprehensibility issues, which include the obscurity of object references, patterns, and overall program structure. Visual debuggers — specifically, ones that display an object diagram to represent the program state — alleviate these issues for imperative programming languages whose very states are directly manipulated by the programmer’s statements. However, these debuggers are also prone to clutter when representing large program states from visualizing too much information. Additionally, the visualized program state can often differ from its conceptual abstraction on paper. We propose *user-defined representation conversion* which allows users to convert concrete representations of program states to their more focused and abstracted conceptual versions. We design a DSL such that users can specify conversions to manipulate displayed nodes and edges based on object types, references, values, or debugger halt locations. We implemented a prototype of this concept for Java.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Integrated and visual development environments*; • **Human-centered computing** → **Information visualization**.

## KEYWORDS

visual debugger, representation conversion, domain-specific language

### ACM Reference Format:

Rifqi Adlan Apriyadi, Hidehiko Masuhara, and Youyou Cong. 2023. Program State Visualizer with User-Defined Representation Conversion (WIP). In *Proceedings of the 1st ACM International Workshop on Future Debugging Techniques (DEBT ’23)*, July 17, 2023, Seattle, WA, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3605155.3605863>

## 1 INTRODUCTION

Displaying program states in graphs instead of the more conventional tree format shows a more accurate representation due to all trees being graphs, yet not necessarily vice versa. For example, in a graph, where each node is an object and each edge is a reference

from one object to another, it is clear which objects have reference to a particular object of interest derived by the source nodes of incoming edges. Additionally, circular references can clearly be shown in a graph in the form of a cycle.

Figure 1 shows a sample subgraph of Java Interactive Visualization Environment’s (JIVE) [6] object diagram view of a Monopoly program that demonstrates these advantages. It is clear that the `StreetProperty` object has two objects that have reference to it signified by the two incoming solid black edges to its node. It is also clear that a circular reference also occurs between it and a `PropertySet`. This information is not clear when using conventional tree-based debuggers.

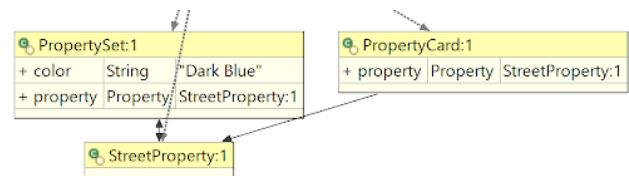


Figure 1: Subgraph of JIVE’s object diagram of a Monopoly program.

Previous visual debuggers [2, 4, 6] utilize the effectiveness of visualizations to help educators teach beginner programmers the concepts of programming. But it could be useful for experienced programmers in debugging their programs as well [10]. The explicitness of information obtained from such visualizations weathers out any obscurity of the program state, allowing for faster behavior comprehension, and therefore, faster bug localization.

However, experienced programmers would find that using these tools to help them debug would be more trouble than it is worth due to the following reasons:

**Visual Clutter** Programs that experienced programmers write typically have a large number of objects and references, consequently adding visual clutter to their representations [5].

**Abstraction Gap** Experienced programmers frequently implement concepts into code differently from how they are commonly imagined — often trading simplicity for efficiency — creating an abstraction gap that lags the translation time from reading the representation on a program level to creating an image of how it looks on a conceptual level [1]. For example, visual debuggers display arrays as a sequence of objects and a developer might need it to be represented as a binary tree for better efficiency.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

DEBT ’23, July 17, 2023, Seattle, WA, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0245-7/23/07...\$15.00

<https://doi.org/10.1145/3605155.3605863>

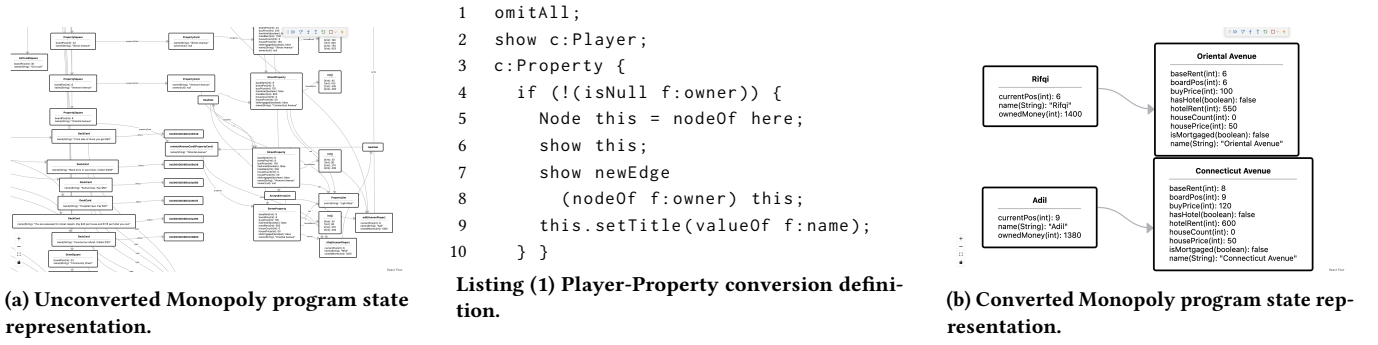


Figure 2: Conversion to only show players and the properties they own.

In large or complex projects, these obstacles ought to be overcome for graph representations of program states to be helpful in accelerating behavior and bug comprehension.

This paper discusses how these problems could be alleviated to preserve the effectiveness of visualizations. Sections 2 and 3 detail the proposed solution and implementation of its prototype, respectively. Section 4 describes the work related to the topic and Section 5 concludes this paper.

## 2 PROPOSAL

This paper proposes the inclusion of user-defined representation conversion functionalities into program state visualizer tools. A dedicated language is provided for users to control the detailed behavior of their conversions. The targets of the conversions are the diagram components of how the program state is represented: nodes, edges, and the text within them. The components and their applicable conversions are:

- Nodes: omission and addition
  - Node Titles: replacement
  - Node Rows: removal and addition
- Edges: omission and addition
  - Edge Labels: replacement

Figure 2 demonstrates this proposal. Figure 2a shows the cluttered program state representation of a Monopoly program without any conversion applied. Figure 2b shows this representation converted to only display nodes of Player objects and the Property objects that they own. This converted representation can be useful when the current bug faced by the programmer concerns the ownership of properties where all other information is irrelevant. Listing 1 is how the programmer would express this conversion. The conversions applied to the representation as defined in Listing 1 are as follows:

- Omit all nodes and edges except for nodes of Player objects and owned Property objects.
- If a Property is owned by a Player:
  - Add and show an edge from the Player owner to the Property.
  - Rename the title of the Property node to the value of its name field for readability.

The inclusion of representation conversion features is hypothesized to solve the problems mentioned in the introduction:

**Visual Clutter** Representations can be converted to omit information irrelevant to the current debug session.

**Abstraction Gap** Representations can be converted to closer resemble the program’s conceptual abstraction (i.e. how the concept looks on paper).

With these obstacles alleviated the advantages of visualization are made available. These include pattern and inconsistency recognition, and more explicit representations, which allow for faster bug identification. For developers new to a codebase, using the visualizer with converted representations should allow for a more comprehensive and intuitive process in understanding the program’s behavior[9].

### 2.1 Usage

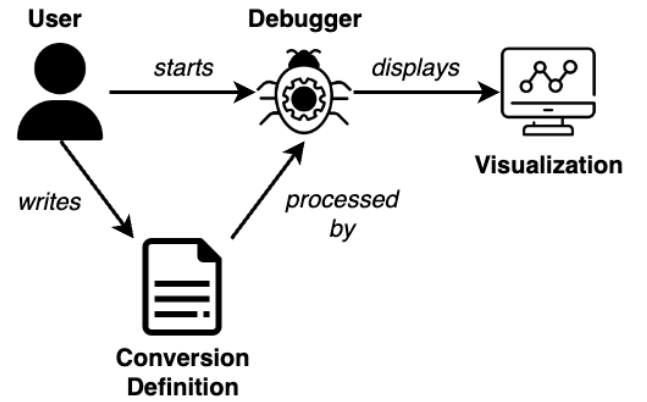
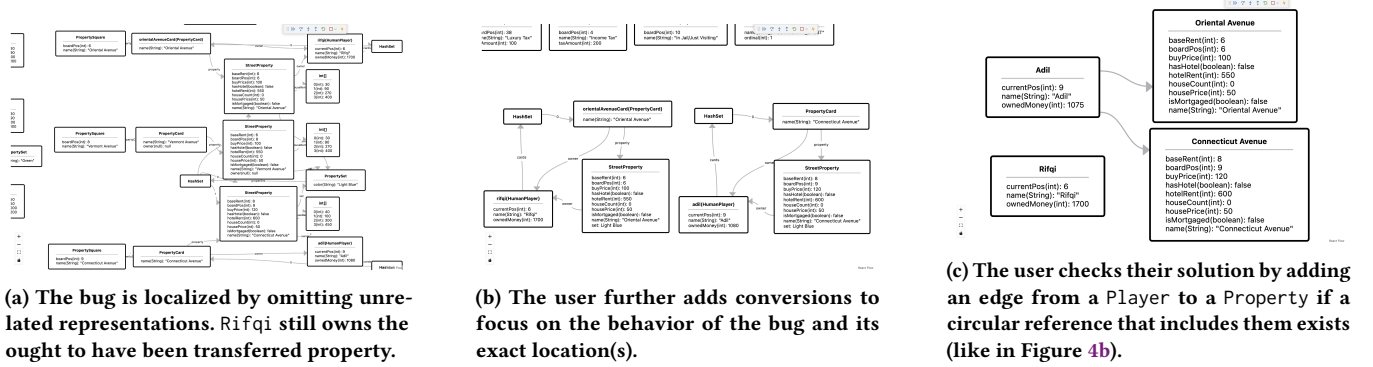


Figure 3: Program State Visualizer usage.

Figure 3 shows how to use the visualizer. The main difference with regular debuggers is the *Conversion Definition*. When the debugger starts, it will process the defined conversion and, combined with the program state data it receives, visualize the converted representation. Changes made to the conversion definition can be reprocessed without restarting the debugger. Doing so would rerender the displayed representation to correspond with the updated conversion definition.



**Figure 4: Incremental representation conversions used in different debugging steps for a bug where property transfer from Rifqi to Adil does not occur when it should.**

In the debugging process, the visualizer and its representation conversion features can be useful in its distinct steps. The conversions would most likely be incremental between each step. The related debugging steps are as follows:

**Bug Localization:** Omitting representations of parts of the code that users are certain are unrelated to the bug can help find the general vicinity of the bug (Example: Figure 4a).

**Cause Identification:** Given a known region of the bug, users can manipulate representations to different levels of detail to pinpoint the specific cause of the bug (Example 4b).

**Solution Implementation:** Users could either reuse their conversions from the previous step to check the correctness of their solution or create a new one if significant changes were made (Example: Figure 4c).

The overhead lies in writing the conversion definitions, which might take time depending on the size and complexity of the program and the nature of the bug. However, they can be shared and reused by multiple users. This problem is elaborated further in Section 2.3 which also describes its solutions.

## 2.2 DSL Design Choices

A design problem in this project is the method by which users specify their conversions. The most prominent factor in this problem is the infinite possible ways users can structure and implement their programs. Furthermore, bugs can appear in any part of their code in any nature which may involve any part of their code. To allow for detailed control, users need a medium with sufficient expressiveness to capture their infinite possible specifications and is focused enough to minimize the overhead in writing them. For these purposes, a domain-specific language (DSL) should serve as a fitting medium [7]. Currently, an external DSL — a language that is parsed independently of the host general-purpose language — is used for the current iteration of the prototype.

This subsection will use Listing 2, which targets a Monopoly program written in Java, as an example conversion definition. For each `StreetProperty` object, it is omitted from the representation if it is unowned. Otherwise, a new edge is made from the owner player node to the property node. Nodes of arrays containing each property's rent prices are also omitted.

```
1 c:StreetProperty {
2     Node thisNode = nodeOf here;
3     if (isNull f:owner) omit thisNode;
4     else {
5         Node ownerNode = nodeOf f:owner;
6         omit edgesOf thisNode ownerNode;
7         show newEdge ownerNode thisNode;
8     }
9
10    f:houseRents {omit nodeOf here;}
11 }
```

**Listing 2: Representation conversion to demonstrate design choices.**

**2.2.1 Imperativeness.** The language is imperative in that the user writes statements that will sequentially be performed, much like writing a new sequential program. This program will be executed each time the debugger halts. Statement types include those that are commonly found in general-purpose imperative languages, such as variable assignments, if-else statements, and while/for loops, and also representation conversion commands, of which show and omit described in Section 2.2.3 are part.

Listing 2 exhibits this property. Lines 2-10 are all done in sequential order. It also resembles something of a program itself. This design decision is the major part of what provides control to the user. The sequential property of the language allows users to control all logic that happens in their specified conversion.

**2.2.2 Subjects.** The DSL has a metaprogramming feature called Subjects which is a type whose objects refer to an object of the target language's running program. A subject can be used to extract the runtime value of an object, its references and referrers, and check whether it is null. It can also be used to get and edit its representation.

In Line 2 of Listing 2, here is a subject expression that refers to the `StreetProperty` object of the current iteration of the encompassing class location and is used to retrieve its node. In Line 3, the `f:owner` subject expression is used to check whether the referred to target is null.

**Table 1: Location types, their semantics, and their declaration restrictions**

Location Type	Block Semantic	Expression Semantic	Declaration Restrictions
Class <i>c</i>	Encompassed statements are applied for all objects of class <i>c</i> .	An array of all runtime objects that are instances of class <i>c</i> .	Blocks: Top-most only Expressions: Unrestricted
Field <i>f</i>	Encompassed statements are applied for all <i>f</i> references of objects of the enclosing class or field block.	The runtime object that is the <i>f</i> reference of the runtime object of the enclosing location block.	In class or field location blocks.
Method <i>m</i>	Encompassed statements are executed when the runtime is currently halted inside method <i>m</i> of some class.	Undefined	Blocks: directly enclosed by a class location block.
Local Variable <i>l</i>	Encompassed statements are applied for the local variable <i>l</i> .	The runtime object with the same variable name in the current scope.	Blocks: Top-most or in method blocks Expressions: Unrestricted

**2.2.3 Nodes, Edges, show, and omit.** The two main components of the diagram subject to conversion are nodes and edges. Without any conversion, the diagram displays all nodes and edges, reflecting the program state. New nodes and edges can be made using the `newNode` and `newEdge` parameterized expressions, respectively. When these expressions are evaluated, a new node or edge is created but is not yet shown.

The two main commands for nodes and edges are `show` and `omit`. When a node or edge is *omitted*, it is removed from the representation. When a node or edge is *shown*, it appears in the representation. Newly created nodes and edges need to be explicitly shown to appear in the representation.

These elements appear in Listing 2 a few times. Lines 3, 6, and 10 omit the node or edges given to them from the representation when they are executed. Line 7 first creates a new edge from the owner player's node to the `StreetProperty` node, then shows it in the representation.

**2.2.4 Locations.** The DSL has a component called *locations* whose role is to represent different components of the target language where conversions can be applied and from which context can be extracted. A location can be written as a block, indicating that encompassed expressions and statements, including other locations, are in its namespace which corresponds to the same namespace in the debuggee program. It can also be written as an expression to refer to the object(s) of the target language that the location represents. In the proof-of-concept made in this research, the types of locations are as seen in Table 1.

Line 1 in Listing 2 means that everything within its curly brackets is executed for every instance of the `StreetProperty` class and its representation. In other words, it is a for-loop that iterates through every `StreetProperty` instance. In each iteration, context regarding the runtime object and its representation is provided. For example, the `here` expression of line 2 refers to the `StreetProperty` object of the current iteration.

The field location block on line 10 is executed in the namespace of the `houseRents` reference of the current `StreetProperty` object of each iteration. Therefore, the `here` on line 10 does not refer to the `StreetProperty` object of the current iteration of the class location block, but rather to its `houseRents` field.

## 2.3 DSL Design Problem and Solutions

Although the design of the language allows users to control conversions with great detail, it also brings with it the problem of being cumbersome to write in. Not only would users need to first learn the language, but they also need to spend the effort to write in it. As seen in the example of Listing 2, 11 lines of code are already required to convey a trivial conversion. It is evident that even more effort would be required to further specify conversions for other parts of the program.

The rest of this subsection discusses current solution concepts to this problem, which have been implemented in the current prototype discussed in Section 3.

**2.3.1 Shortcuts.** An intuitive solution to the cost overhead in defining conversions is to provide "shortcuts" to reduce it. Here, *shortcuts* in the DSL refer to function-like constructs that perform a sequence of conversions. The idea is that commonly written conversions or those that will be written multiple times in different parts of the specification should be trivialized.

For example, the DSL has a predefined shortcut called `inline` that takes a location expression as its argument which does the following:

- Omits the node(s) of the given location.
- For each object that has a reference to a structured object *j* corresponding to the location, add a row in its node that displays the `toString()` of *j*.

Figure 5 shows this in action. Assume that the `toString()` value of a `PropertySet` is its `color` value. With Figure 5a showing the unconverted representation, including `inline c:PropertySet;` would convert it to look like the representation in Figure 5b. Note that the location expression was `c:PropertySet`, meaning that this conversion applies to representations of all other `PropertySet` instances as well.

Intuitively, this may relieve the costs of writing in the language somewhat. To further the theme of being a function-like construct, shortcuts ought to also be user-defined on top of the predefined ones. However, doing so would significantly steepen the learning



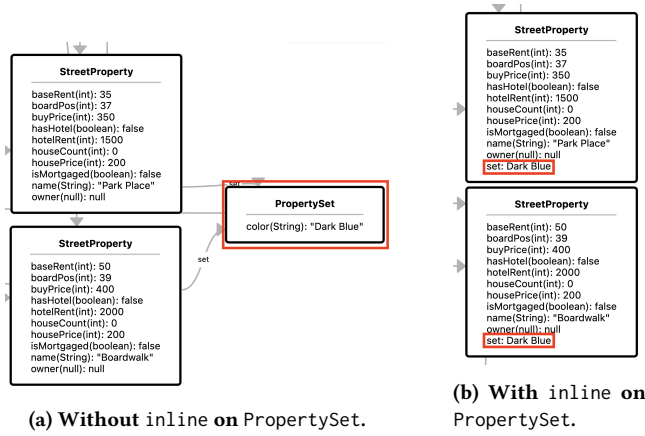


Figure 5: Plain and inlined `PropertySet`.

curve, making it more expensive to learn the language. Fortunately, there might be a solution for this still.

**3.3.2 Internal DSL.** Switching to an internal DSL in a general-purpose programming language ought to reduce the learning costs of the language given the familiarity of the host language, especially if the host language is the same as the debugger’s target language. Furthermore, it would also reduce the costs of writing definitions as it would open users to the existing capabilities of the host language and IDEs that support that language.

The original reason to provide an external DSL was that the language was intended to be focused, providing only a small range of necessary features. However, further iterations of the language’s development showed that the language continued to closer resemble a general-purpose language, especially with the addition of custom shortcuts. Furthermore, one of the main purposes of an external DSL is as an intermediate language between experts of the problem domain and developers; which in this case are the same people.

**3.3.3 Implicit Statements.** The current design of the DSL explicitly separates the metaprogramming features from the representation conversion parts of the language. This is most evident in how subjects are separated from their representation properties. This separation causes users to write separate statements for context and conversion.

Blurring this separation and making implicit connections between the target program with its representation should streamline defining conversions and make the process more intuitive. A draft example: given a `PropertyCard` object `pc` with a reference to a `Property` object `p`, writing `pc.p.title = pc.p.name`; would change `p`’s node title to be the value of the name string of `p`.

### 3 IMPLEMENTATION

Our prototype called *JIGSAW*<sup>1</sup> has been implemented for Java for its object orientation and because it is the language of choice for most related works. *JIGSAW* is implemented as a Visual Studio Code (VSCode) extension due to VSCode’s extensive debugging tooling support.

<sup>1</sup><https://github.com/adilrifqi/jigsaw>

*JIGSAW* has a backend and a frontend. The backend is responsible for parsing debugging data and conversion definitions and generating a collection of nodes and edges to be sent to the frontend. The frontend is responsible for creating the node and edge objects, their layout, and their visualization.

Upon startup, the backend’s parser parses and type-checks the conversion definition the user wrote to create an internal *Conversion Model*. The current external DSL was designed and written using ANTLR<sup>2</sup>, which provides the parser.

On the other hand, in every debugger halt, messages fired between VSCode and Java’s language support extension via the Debug Adapter Protocol (DAP)<sup>3</sup> are tracked by the backend to create an internal *Debug State Model*.

These two internal models are used to first generate a collection of nodes and edges that mirrors the program state without conversions. This is then updated using the conversion model, essentially running the code written in the conversion definition. The resulting nodes and edges are sent to the frontend. Reprocessing a conversion definition while the debugger is still running follows the same process.

The frontend, essentially the Webview part of *JIGSAW*, simply receives the data of the nodes and edges that need to be shown, builds them, lays them out, and displays them. The user interface uses React Flow<sup>4</sup> to draw graphs. *elkjs*<sup>5</sup> is used to lay out the nodes and edges of the graph.

The graph view currently has limited capabilities. Users can zoom in and out of the view and move nodes around. However, features that would further improve the debugging experience, such as text finding, node location retention between steps, and interactive representation conversion are not yet present.

Although the target language of the implemented prototype is Java, the concept discussed in this work is applicable to all imperative languages, with slight differences corresponding to language-unique features.

## 4 RELATED WORK

Torchiano [9] conducted an experiment to verify whether using UML diagrams improves program system comprehension. The experiment involved two groups given the same set of tests with opposing permissions to use object diagrams for each test. Using the standardized effect size, three out of four tests showed small or medium-sized effects, implying the correctness of the hypothesis. His later work [10] involves a family of four controlled experiments to assess whether the use of UML object diagrams improves the comprehension of program design when added to UML class diagrams. The results showed that this is only mostly true for more experienced programmers. It implied that programming experience and UML familiarity should be considered in using object diagrams for software modeling in program design comprehension.

Java Interactive Visualization Environment (JIVE) [6] offers interactive features to display how a Java program runs at various levels of detail. Among its features is a program state visualizer in the form of an object diagram. JIVE’s solution to managing large

<sup>2</sup><https://www.antlr.org/>

<sup>3</sup><https://microsoft.github.io/debug-adapter-protocol/>

<sup>4</sup><https://reactflow.dev/>

<sup>5</sup><https://github.com/kieler/elkjs>

executions is class exclusion filters, debugging an interval of code, and focusing on a specific object. Though they may somewhat resolve visual clutter, it does not have the granularity and control our solution provides. With the proposed language, these solutions can also be defined along with other conversions the user might need.

BlueJ [2] is a tool that is specifically designed for instructing individuals on the principles of object-oriented programming using the Java language. A "class view" feature enables the visualization of the interconnections between classes, while the "object dock" displays all initialized objects. JAVAVIS [4] is a tool that aids students in comprehending Java programs through the use of dynamic object and sequence diagrams that depict program executions. Though both of these tools provide different types of views to understand different aspects of the program, they do not have features for visual scalability as their intended use is for small and simple programs frequently made for educational purposes.

Velázquez-Iturbide [11] proposed the integration of visualizations customization into WinHIPE [8], which is the Windows version of a programming environment for the functional programming language Hope [3]. This feature allows the programmer to customize the visualization of intermediate expressions resulting during any evaluation. Customizations allow users to choose text vs. graphics and typographic styles and simplify visualizations using "fisheye views". This was a step closer to faster program comprehension and better visualization readability. Our work has a similar goal for imperative languages where visualizations are of the program state. The proposed representation conversion is essentially customizing what and how information is displayed, only on a more detailed and larger scale.

## 5 CONCLUSION AND FUTURE WORK

Visualization of program states is beneficial for programmers new and experienced alike. However, visual clutter and abstraction gaps diminish its effectiveness the larger or more complex a program grows. These obstacles can be overcome with the ability to convert representations in the visualization and can be useful in most parts of the debugging process. For this to be effective, users ought to be able to granularly control the behavior of their conversions.

Users can define their conversions using a DSL. The DSL uses an imperative design and can retrieve values of the target language. However, this design creates a large overhead in writing conversion definitions. Furthermore, continued iterations of the language increased the concepts new users would need to learn to write in the language. To solve these problems, shortcuts and an internal version of the DSL ought to minimize both the learning costs and definitions costs.

A prototype of this concept called JIGSAW has been implemented with Java as the target language of the debugger as an extension to VSCode. It uses information tracked via the DAP to extract a Debug State Model and from the user's conversion definition to build Debug State and Conversion Models that are together used to build the visualized representation. Although JIGSAW has been built for Java, the concept is language-agnostic.

Currently, the migration to using an internal DSL from an external one remains only an idea. Future work consists of implementing it while also changing its design to provide more streamlined implicit connections between the debuggee with its representation. It also consists of implementing the missing features mentioned in Section 3. Additionally, we also plan on designing and conducting evaluations on the effectiveness of the system proposed in this research, which would probably be in the form of user tests.

## REFERENCES

- [1] Fahad Alhumaidan and Nazir Ahmad Zafar. 2014. Possible improvements in UML behavior diagrams. In *2014 International Conference on Computational Science and Computational Intelligence*, Vol. 2. IEEE, 173–178. <https://doi.org/10.1109/CSCIL2014.113>
- [2] Jens Bennedsen and Carsten Schulte. 2010. BlueJ visual debugger for learning the execution of object-oriented programs? *ACM Transactions on Computing Education (TOCE)* 10, 2 (2010), 1–22. <https://doi.org/10.1145/1789934.1789938>
- [3] Rod M Burstall, David B MacQueen, and Donald T Sannella. 1980. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*. 136–143. <https://doi.org/10.1145/800087.802799>
- [4] Miguel Cazorla and Diego Viejo. 2015. JavaVis: An integrated computer vision library for teaching computer vision. *Computer Applications in Engineering Education* 23, 2 (2015), 258–267. <https://doi.org/10.1002/cae.21594>
- [5] Lukas Holy, Kamil Jezek, Jaroslav Snajberk, and Premek Brada. 2012. Lowering visual clutter in large component diagrams. In *2012 16th International Conference on Information Visualisation*. IEEE, 36–41. <https://doi.org/10.1109/IV.2012.17>
- [6] Demian Lessa, Jeffrey K Cxyz, and Bharat Jayaraman. 2010. JIVE: A pedagogic tool for visualizing the execution of Java programs. *Bericht, Univ. of New York, Buffalo* (2010).
- [7] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344. <https://doi.org/10.1145/1118890.1118892>
- [8] Cristóbal Pareja-Flores, Jamie Urquiza-Fuentes, and J Angel Velázquez-Iturbide. 2007. WinHIPE: An IDE for functional programming based on rewriting and visualization. *ACM SIGPLAN Notices* 42, 3 (2007), 14–23. <https://doi.org/10.1145/1273039.1273042>
- [9] Marco Torchiano. 2004. Empirical assessment of UML static object diagrams. In *Proceedings, 12th IEEE International Workshop on Program Comprehension, 2004*. IEEE, 226–230. <https://doi.org/10.1109/WPC.2004.1311064>
- [10] Marco Torchiano, Giuseppe Scanniello, Filippo Ricca, Gianna Reggio, and Maurizio Leotta. 2017. Do UML object diagrams affect design comprehensibility? Results from a family of four controlled experiments. *Journal of Visual Languages & Computing* 41 (2017), 10–21. <https://doi.org/10.1016/j.jvlc.2017.06.002>
- [11] JÁ Velázquez-Iturbide and Antonio Presa-Vázquez. 1999. Customization of visualizations in a functional programming environment. In *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011, Vol. 2. IEEE, 12B3–22*. <https://doi.org/10.1109/FIE.1999.841580>

Received 2023-05-22; accepted 2023-06-20