TOKYO INSTITUTE OF TECHNOLOGY

MASTER'S THESIS

Adding User-Definable Representation Conversion to Debugger State Visualization

Author:
Rifqi Adlan APRIYADI

Student Number:
21M38030

Supervisor:
Prof. Hidehiko MASUHARA

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Programming Research Group
Department of Mathematical and Computing Science

August 29, 2023

Declaration of Authorship

I, Rifqi Adlan APRIYADI, declare that this thesis titled, "Adding User-Definable Representation Conversion to Debugger State Visualization" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Debugging is like being the detective in a crime movie where you're also the murderer."

Filipe Fortes

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Department of Mathematical and Computing Science

Master of Science

Adding User-Definable Representation Conversion to Debugger State Visualization

by Rifqi Adlan APRIYADI

Trees, the most commonly used data structure to display debugger states for imperative programming languages, have issues where the accuracy of the debugger state shown is limited by trees' properties. Because graphs have a non-hierarchical structure and are also the superset of trees, graphical visual debuggers alleviate these issues by allowing them to show debugger state information that cannot be shown in trees.

In doing so, however, they become prone to visual clutter when representing larger debugger states due to the increase in objects and references to display. Additionally, gaps in abstraction are likely to occur between the debugger state representation and its conceptual abstraction on paper. Both these problems hinder the effectiveness of using visualizations and instead make behavior comprehension more unwieldy.

This study proposes user-definable representation conversion which allows users to convert concrete representations of debugger states to their more focused and abstracted conceptual versions. It is believed that this feature could mitigate the two previously mentioned problems by allowing users to choose which and how information is displayed, allowing them to focus on information relevant to their current circumstances. It is also believed that this feature can assist users in distinct steps of the debugging process. To provide control and versatility, a domain-specific language is included in the feature with which users can specify conversions to manipulate displayed nodes and edges based on object types, references, values, or debugger halt locations. A prototype for this concept has also been developed to help debug Java programs.

Acknowledgements

I would like to express my gratitude to the individuals and groups mentioned in this section who have been instrumental in the successful completion of my thesis and studies.

First and foremost I would like to show appreciation to my academic supervisor, Professor Hidehiko Masuhara, for his unwavering support and patience throughout this journey. His expertise and mentorship have guided me to move forward in the important directions I needed to take. I also appreciate the freedom he has given me in doing this research, including the topic I chose, the approaches I took, and determining the depth and significance to which I take this research. Additionally, I value that he has encouraged the use of English in the lab, without which I would have been lost many times over. I would also extend this appreciation to Professor Youyou Cong, whose insightful inquiries, perspectives, and critiques have further enriched my academic journey.

I would also like to thank members of the lab from which I have taken much inspiration. Their perception and feedback helped me in making decisions over the course of my studies. I especially appreciate the help from Rikito Taniguchi whose experience with the necessary tools aided me in the development of the prototype built in this study.

Closely intertwined with my academic journey, I would like to extend my appreciation to the Tokyo Tech International Student Association (TISA) council members, with whom I have spent a considerable amount of time organizing and attending events. I am truly grateful for the friendships cultivated and the shared experiences within this group.

Finally, I would like to extend my heartfelt gratitude to my family, whose constant encouragement, understanding, and belief in my abilities have provided me with the foundation to overcome challenges and pursue excellence. Thank you, Kak, for your presence. Thank you, Pa, for your guardianship. Thank you, Ma, for your faith.

Contents

List of Figures

List of Listings

List of Tables

List of Abbreviations

UDRC	**U**ser-**D**efinable **R**epresentation **C**onversion
CD	**C**onversion **D**efinition
TBD	**T**ree-**B**ased **D**ebugger
JIVE	**J**ava **I**nteractive **V**isualization **E**nvironment
VSCode	**V**isual **S**tudio **Code**
DSL	**D**omain-**S**pecific **L**anguage
DGV	**D**ynamic **G**raph *V*isualization
DAP	**D**ebug **A**dapter **P**rotocol
UI	**U**ser **I**nterface
GUI	**G**raphical **U**ser **I**nterface

Chapter 1

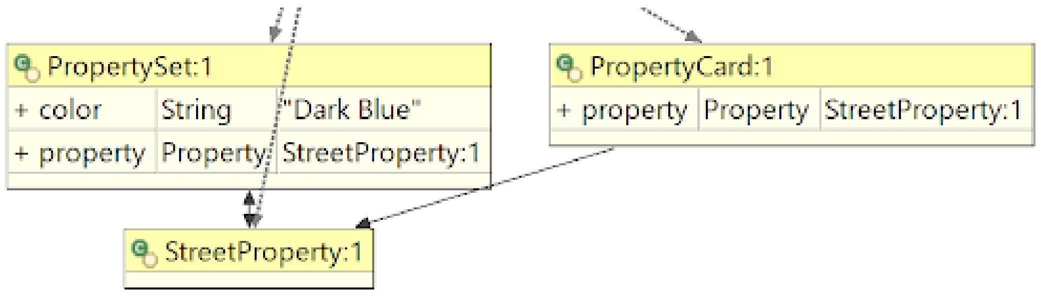
Introduction

Conventional debuggers for imperative programming languages — where debugger states are directly manipulated by the programmer's statements — show these states by means of a tree. Each node represents a runtime object, an object of the running debuggee, and expanding the node reveals the nodes of the object's referenced objects. This allows for a hierarchical representation of variables.

However, runtime objects and references are not hierarchical as different objects can have references to the same object, and circular references can occur. Analogously, each node in trees can only have at most one parent and cannot have a child that is also its ancestor. Multiple nodes need to be shown for the same runtime object if that object is referenced by multiple others. Furthermore, the method by which these debuggers show circular references is to allow the infinite expansion of nodes if the nodes of the objects in the circular reference continue to be expanded. These flaws illustrate the counterintuitiveness of using a tree structure to display debugger states.

Changing the structure used in displaying the debugger state is an intuitive solution to solve this problem. Graphical visual debuggers, which show variables by representing runtime objects as nodes and references between objects as edges between their nodes, display graphs to represent the debugger state as opposed to trees. Graphs can show debugger states non-hierarchically and, due to being the superset of trees, can show information trees cannot.

Figure 1.1 shows a sample subgraph of Java Interactive Visualization Environment's (JIVE) [Lessa, Czyz, and Jayaraman, 2010] object diagram view of a Monopoly program. In this subgraph, there are three runtime objects, each an instance of different classes: `PropertySet`, `PropertyCard`, and `StreetProperty`. A solid black edge represents the object of the source node having reference to the object of the target node. For example, the `PropertyCard` object has a reference to the `StreetProperty` object.

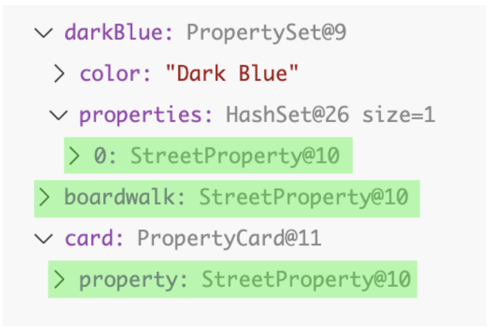


PropertySet:1
+ color | String | "Dark Blue"
+ property | Property | StreetProperty:1
PropertyCard:1
+ property | Property | StreetProperty:1
StreetProperty:1

FIGURE 1.1: Subgraph of JIVE's object diagram of a Monopoly program.

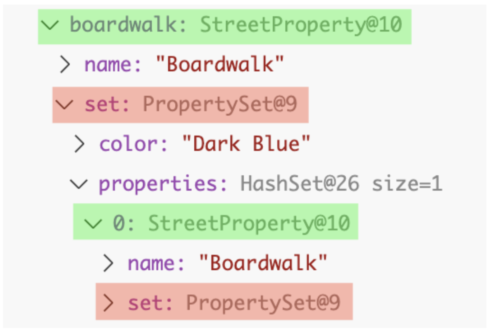
With graphs, multiple edges can target the same node, meaning that each object can be represented by only one node, and referencing objects can be derived by the source nodes of incoming edges. Circular references are also displayed accurately due to graphs being allowed to have cycles. The two-dimensionality of a graph as a visualization also better displays the debugger state, further enhancing the intuitiveness of the displayed information.

In Figure 1.1, the `StreetProperty` object's node has two incoming edges — representing that there are two objects with reference to itself — while retaining only one node instead of adding a node for each reference to it. The `StreetProperty` object also has a circular reference with the `PropertySet` object indicated by the two directed edges between their nodes targeting each other. Figure 1.2 — taken from Visual Studio Code (VSCode)[1] — shows how the same debugger state is shown in a tree-based debugger (TBD) where the same highlight color indicates the same runtime object.



```
∨ darkBlue: PropertySet@9
  > color: "Dark Blue"
  ∨ properties: HashSet@26 size=1
    > 0: StreetProperty@10
> boardwalk: StreetProperty@10
∨ card: PropertyCard@11
  > property: StreetProperty@10
```

(A) Object `StreetProperty@10` has multiple nodes from multiple incoming references.



```
∨ boardwalk: StreetProperty@10
  > name: "Boardwalk"
  ∨ set: PropertySet@9
    > color: "Dark Blue"
    ∨ properties: HashSet@26 size=1
      ∨ 0: StreetProperty@10
        > name: "Boardwalk"
        > set: PropertySet@9
```

(B) Objects `StreetProperty@10` and `PropertySet@9` have a circular reference, allowing infinite expanding descent.

FIGURE 1.2: The TBD version of Figure 1.1.

Previous visual debuggers utilized this type of visualization to assist educators in teaching beginner programmers the behavior of objects and references corresponding to the statements they wrote. They allow beginners to more intuitively extract the debugger state by showing what TBDs cannot. For example, the fact that each runtime object corresponds to only one node despite the number of objects referencing it gives a more correct intuition of how references work. This is especially helpful to teach the concept of reference semantics of different programming languages. On the other hand, representing the same information in TBDs where multiple nodes are displayed to represent the same object gives the opposite intuition that they are different objects, as demonstrated in Figure 1.2a.

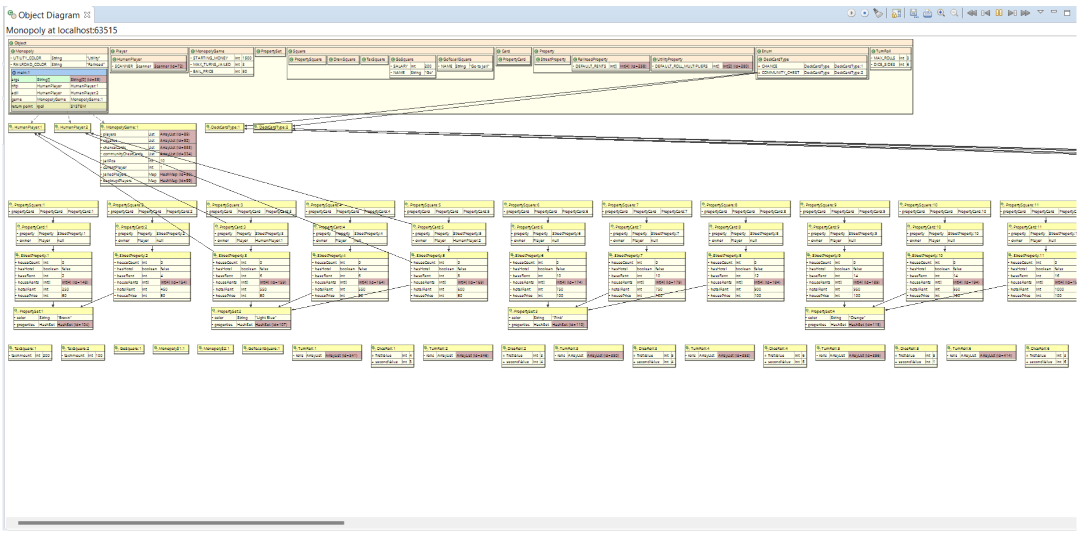
Additionally, graphical visual debuggers could also be helpful to experienced programmers in finding bugs [Torchiano et al., 2017]. Though experienced programmers are likely to already understand the concept of objects, references, and reference semantics, its explicit visualization helps in interpreting it. In contrast, TBDs do not make this information as accessible. Moreover, the general advantages of visualization are also present, including pattern and inconsistency recognition, which allow for faster bug identification.

However, experienced programmers might not feel as though these advantages are provided to them and that using these tools for visualization is more trouble than it is worth due to the following reasons:

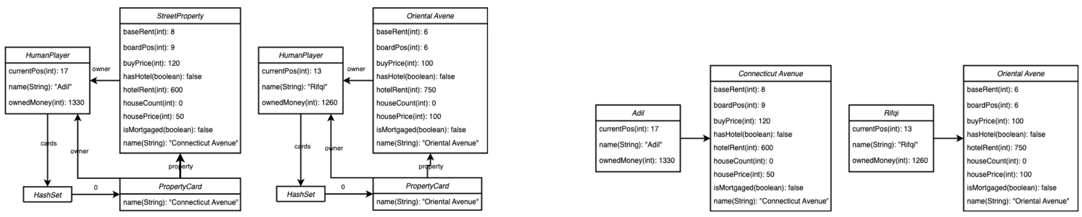
[1]https://code.visualstudio.com/

Visual Clutter Programs that experienced programmers write are typically sizeable with a large number of objects and references, consequently adding visual clutter to their representations [Holy et al., 2012].

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 [Alhumaidan and Zafar, 2014]. For example, visual debuggers display arrays as a sequence of objects and a developer might need it to be represented as a binary tree.



(A) Visual clutter in displaying the Monopoly program using JIVE.



(B) Concrete debugger state representation after filtering (left) vs. how the programmer would like it being represented (right).

FIGURE 1.3: Examples of visual clutter and abstraction gap in the Monopoly program.

Figure 1.3 shows examples of these problems. The full representation of the Monopoly program can be seen in Figure 1.3a where nodes and edges are in abundance, causing clutter and difficulty in understanding the debugger state. But even if the representation can be filtered to show only certain parts of the state, how the information is presented remains unchanged. Figure 1.3b shows that even after using a sort of filter feature, there is still a gap between how the concrete representation of the debugger state looks and how the programmer imagines its information. The left part of the figure remains a one-to-one representation of that specific part of the debugger state. This creates a lag from reading the representation to extracting the information users actually need.

In large or complex projects, these hindrances ought to be overcome for the above-mentioned benefits to be accessible through the graph representations of debugger states in the visualization and help in accelerating behavior and bug comprehension.

This thesis presents a novel feature for debugger state visualizers that mitigates the detriments of size and complexity of debugged programs in their graphical representations. Chapter 2 details the design and intended benefits of the feature, along with the external domain-specific language (DSL) users use to control the feature. Chapter 3 describes the prototype made in this study, its architecture, and its implementation. Chapter 4 evaluates the feature's hypothetical effectiveness through a comprehensive exploration of a fictional case study. Chapter 5 discusses prior work done with the same goal or the same methods, as well as work that supports ideas of graphical visualizations of debugger states. Chapters 6 and 7 address future work that can be done for this research and the outlined summary of this research, respectively. Finally, Appendix A contains the full grammar of the DSL that has been designed and implemented in this study.

Chapter 2

Proposal

2.1 Overview

2.1.1 *User-Definable Representation Conversion* (UDRC)

To solve the problem mentioned in Chapter 1, the addition of *user-definable representation conversion* (UDRC) to debugger state visualizer tools is proposed. This feature would allow users to convert the structure of the debugger state representation — in other words, altering information in nodes and edges — by defining the conversion's behavior. In other words, the same debugger state can be made to be represented in a different manner with a differently structured object diagram. The purpose of this feature is to assist the comprehension of the debugger state by altering what and how the debugger state representation is displayed to allow users to focus on information relevant to their current circumstances.

To define the behavior of the representation conversion, a DSL is provided that users can write with. The DSL allows users to imperatively declare behavior with commonly-used programming language constructs. The DSL would also provide metaprogramming features that supply information regarding the debuggee for context when applying conversions.

In terms of what users can do to the displayed representation, all structural parts of the graph can be customized. The graph 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

For example, Figure 2.1 is a screenshot of the unconverted graphical representation of a Monopoly program's current debugger state. In other words, the nodes correspond one-to-one with non-string non-primitive objects and the edges also correspond one-to-one with all references in the current debugger state. If currently the user is faced with a bug relating to which *Player* owns which of the game's *Properties*, then all other information that the user is certain does not concern the issue can be considered visual noise.

To eliminate the noise, the user can define the conversion of this representation as in Listing 2.1. It should be noted that in this particular Monopoly program, *Players* do not have a set of owned properties but instead have a set of owned *Cards*, most of

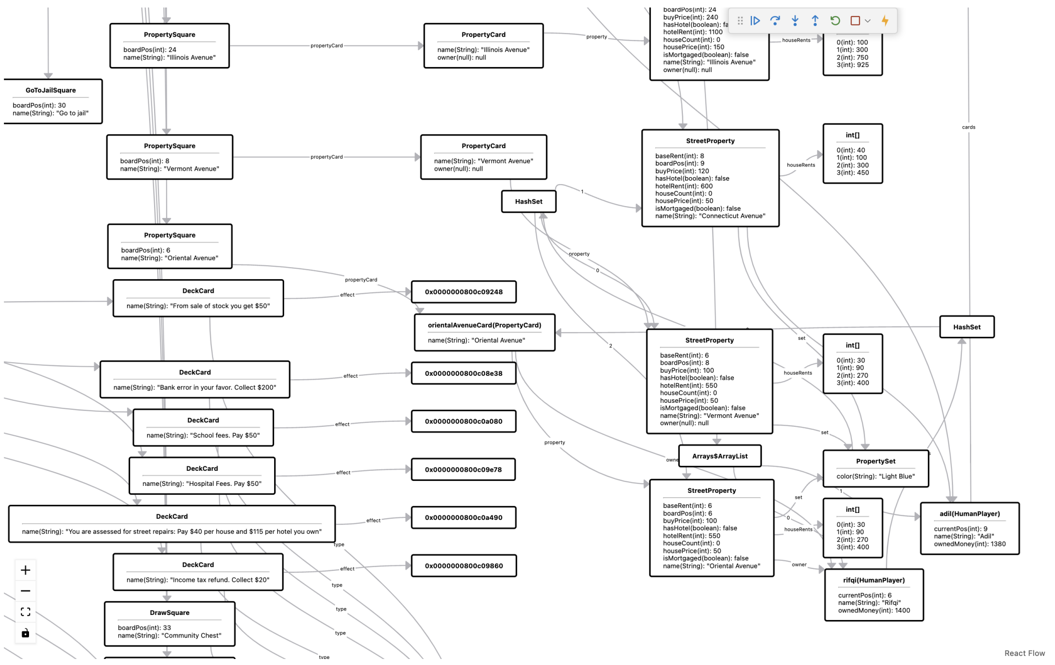


FIGURE 2.1: Unconverted Monopoly debugger state representation.

```
1  omitAll;
2  show c:Player;
3  show c:Player.f:cards;
4  show childrenOf c:Player.f:cards;
5  show (childrenOf c:Player.f:cards).f:property;
```

LISTING 2.1: CD to remove clutter from Figure 2.1

which are *PropertyCards*, which are associated to *Properties*. For the sake of simplicity, this conversion assumes that all owned cards are *PropertyCards*. The behavior declared in this conversion definition (CD) line-by-line is as follows:

1. Omit all nodes from the graph (ergo all edges as well)
2. Show all *Players'* node representations
3. Show the nodes representing the sets containing these players' cards
4. Show the nodes representing references made by these sets (i.e. the cards contained in the sets)
5. Show the nodes representing the `property` fields of these cards (which are assumed to be *PropertyCards*).

The representation of the same debugger state represented in Figure 2.1 converted using the definition in Listing 2.1 is shown in Figure 2.2. The visual clutter from the previous representation has been omitted from the view, leaving the user with only the relevant parts.

However, perhaps the user would find it easier to read this debugger state if it was represented differently: by further converting it to represent the same amount of information in a different manner. In other words, the next step would be to convert

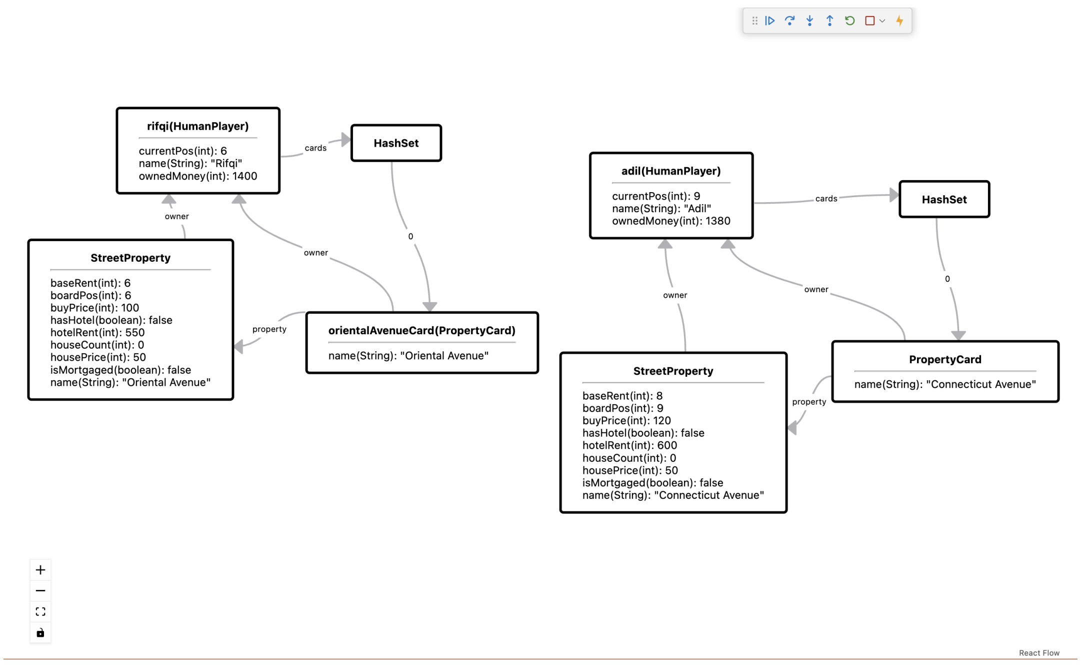


FIGURE 2.2: Uncluttered representation of Figure 2.1 using the conversion defined in Listing 2.1.

how it is represented instead of *which* of it is. For example, using the CD in Listing 2.2 which converts the representation of the same debugger state to resemble that of Figure 2.3.

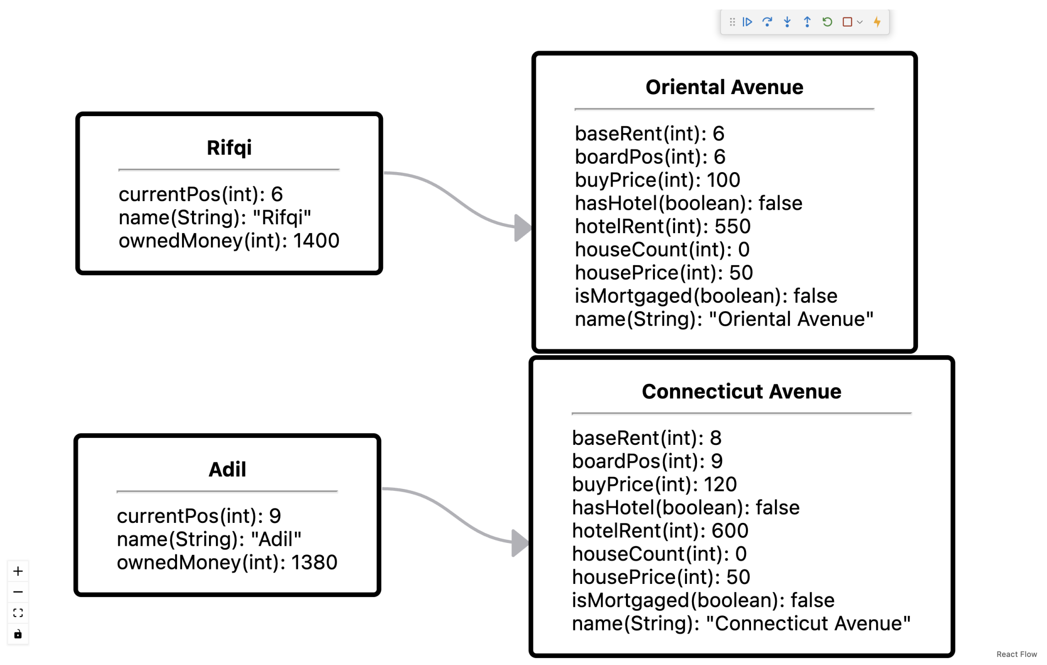
Listing 2.2 further converts the representation in Figure 2.2 by replacing the conversion defined in Listing 2.1 and do the following:

1. Showing all *Player* nodes and changing their titles to the names of the players (Lines 2 - 5).

2. If a *Player* has an owner (Line 7), show its node (Line 11), a new edge from the player to this node (Line 12), omit the opposite edge (Line 13), and set the title of the *Property* node to the name of that property.

The representation now only shows nodes of players and the properties that they own with direct edges from the former to the latter. Although this does not accurately represent the concrete debugger state in terms of the actual references that exist, users can better comprehend the information regarding "who owns which properties" in each state. If the user is faced with a bug concerning this information, this is one possible representation the user might arrive at.

Given how representation conversion works and what it can be used for, the concept is language-agnostic for imperative programming languages. In the imperative programming paradigm, users write statements that are executed one after another that change the program's state. Visualization of these states in debugging is merely a more enhanced way of displaying debugger states compared to TBDs, whose concept is itself language-agnostic. Therefore, the feature of UDRC is only as language-agnostic as the visualizer it is a part of.



Rifqi
currentPos(int): 6
name(String): "Rifqi"
ownedMoney(int): 1400

Oriental Avenue
baseRent(int): 6
boardPos(int): 6
buyPrice(int): 100
hasHotel(boolean): false
hotelRent(int): 550
houseCount(int): 0
housePrice(int): 50
isMortgaged(boolean): false
name(String): "Oriental Avenue"

Adil
currentPos(int): 9
name(String): "Adil"
ownedMoney(int): 1380

Connecticut Avenue
baseRent(int): 8
boardPos(int): 9
buyPrice(int): 120
hasHotel(boolean): false
hotelRent(int): 600
houseCount(int): 0
housePrice(int): 50
isMortgaged(boolean): false
name(String): "Connecticut Avenue"

FIGURE 2.3: A more readable representation of that shown in Figure 2.2.

2.1.2 Representation Conversion Effects

The goal of UDRC is to alleviate the obstacles mentioned in Chapter 1, as also exemplified by the above example:

Visual Clutter Representations can be converted to omit information irrelevant to the current bug.

Abstraction Gap Representations can be converted to portray different levels of abstraction of the same information. This can be to make it more readable and/or to closer resemble the program's conceptual abstraction on paper (i.e. how the concept looks on paper).

With these obstacles accounted for, the advantages of visualizations are once more made available to the user. They include pattern and inconsistency recognition [Sadiku et al., 2016], and more explicit representations, which allow for faster bug identification [Strobelt et al., 2018]. The behavior of the program with regard to the debugger state can also clearly be seen between steps, especially when it involves reference semantics and changes in references. 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 [Torchiano, 2004].

A design problem of this feature is the way in which users define their conversions to control their behaviors. The biggest part of this problem is the endlessness of possible scenarios that users can face. There are seemingly infinite concepts or tasks that can be programmed, each with infinite approaches that can be taken for its design. In a particular program, bugs can appear anywhere in any nature, whose visible side-effects can be anywhere in the program depending on the chosen design. To solve these bugs, various possible parts of the code can be relevant information with different possible ways they could be represented.

```
omitAll;
c:Player {
    show here;
    (nodeOf here).setTitle(valueOf f:name);
}
c:Property {
    if (!(isNull f:owner)) {
        Node propNode = nodeOf here;
        Node ownerNode = nodeOf f:owner;

        show propNode;
        show newEdge ownerNode propNode;
        omit edgesOf propNode ownerNode;
        propNode.setTitle(valueOf f:name);
    }
}
```

LISTING 2.2: CD to make Figure 2.2 more readable.

For example, from an infinite number of possible concepts to write programs of, one may choose to write a program for a board game. This choice already implies that the program would probably have one central *Board* and/or *Game* object that keeps track of relevant objects and dictates the flow of the game. Other parts of the program can be differently designed. *Properties* can be separated from their *Cards*, or one may exist but not the other. Game actions can have their own classes or not at all. Additionally, a bug can occur caused by any part of the code. If a bug where a **Player** that owns a full *PropertySet* is not detected correctly, it can cause problems in the *Property* and/or *Card* classes in terms of mortgaging, buying houses, and other players paying rent. It can also have other impacts depending on the chosen design of the program.

2.1.3 User Control

Considering all these possibilities, users need a medium with sufficient expressiveness to capture even the small details of their infinitely possible CDs and one that is focused enough to minimize its learning and usage overhead. For this purpose, a DSL ought to fit the description [Mernik, Heering, and Sloane, 2005] for these reasons:

Expressiveness DSLs can be designed to have as much expressiveness as it needs. In this case, the DSL would need to be capable of expressing different object types, metaprogramming features and values, representation conversion statements, and constructs that bridge them all together.

Focus DSLs can also be designed to not have features they do not need, hence the name *Domain-Specific* Language: They should be designed to fulfill specific purposes. In this case, the features present in the DSL should only be those necessary to view the debuggee's debugger state and convert its representation. For example, concurrency would not be a necessary feature for representation conversion.

Users need fine-grained control over the behavior of their conversions so that they could represent their debugger states as necessary. This detailedness goes as far as defining conversions statement-by-statement in the hopes to provide the most conversions definable to accommodate virtually infinite program possibilities.

Although providing user control through the graphical user interface (GUI) would be an intuitive approach for a tool like this, it was not deemed sufficient. Simple conversions, which are mostly those that mitigate visual clutter, such as omitting all nodes of instances of a class, can be handled by GUI controls. However, more complicated conversions, which are mostly occupied by those that alleviate abstraction gaps, cannot as easily be captured. This is because CDs in the latter type are more complex with sequential behaviors.

To be able to accommodate this, the GUI would have predefined conversions or have a feature that allows users to define their own conversions. The former has already been attempted in previous works [Lessa, Czyz, and Jayaraman, 2010, Bennedsen and Schulte, 2010, Cazorla and Viejo, 2015] but they have proven that they are not sufficient. If those features proved inadequate, one might question the efficacy of incorporating more, as more possible requirements may arise. Implementing the latter, however, would merely make the GUI a more cumbersome way to write statements in this DSL.

For example, Listing 2.1 is a definition that alleviates visual clutter by omitting everything but a few parts of the state. This can be easily done with a GUI control feature. However, Listing 2.2 is more complex with an if-statement. It does not stop other conversions to have more complex conditional branches or loops. With a GUI, it would have to be able to capture these branches and loops, making it eerily similar to writing a program.

2.2 Usage

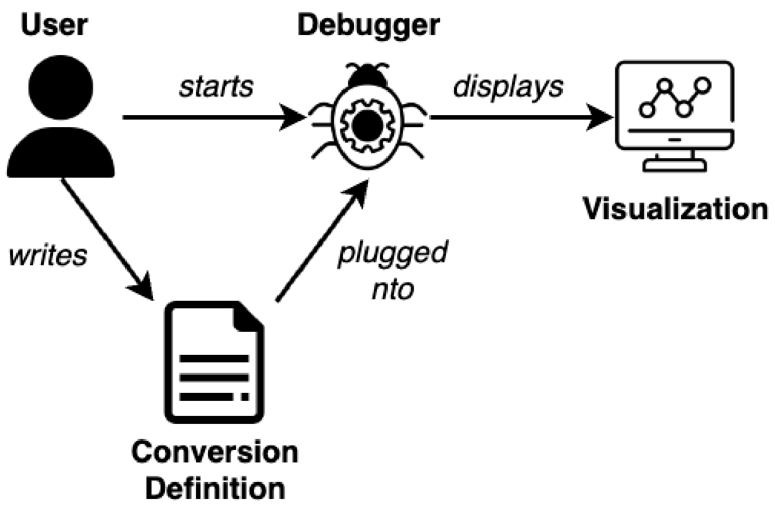


FIGURE 2.4: Usage of the visualizer.

Figure 2.4 depicts how the visualizer can be used. The top part shows how conventional debuggers are normally used: the user runs the debuggee in debug mode and the debugger shows the debugger state information once it hits a breakpoint using a TBD or graphical representation. With this visualizer, the difference is in the additional step of writing a CD. When the debugger is started, it reads and compiles the CD which, combined with the debugger state data of the debuggee, displays a converted representation of the state. Changes made to the CD can be reprocessed in the middle of a debug session without having to restart the debugger. Doing so would rerender the displayed representation to correspond with the updated CD.

Using a previous example, Figure 2.1 is a representation of a debugger state of a Monopoly program with an empty CD, meaning no conversion is applied to the representation of the debugger state. Without restarting the debugger, the user writes a CD as in Listing 2.1 based on their needs and triggers a recompilation of the used definition, rerendering the same debugger state to look like Figure 2.2.

As an added benefit, considering that a CD is only a file, it becomes readily shareable with colleagues, students, or online platforms, thereby promoting collaborative use and knowledge dissemination. Consequently, this would significantly curtail the resources allocated to writing CDs.

2.2.1 Practicality in Debugging

In the debugging process, representation conversion can be useful in its steps that involve behavior and bug comprehension. The conversion would most likely be incremental between steps, which is supported by the tool by facilitating the reprocessing of CD changes without restarting the debugger. 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 identify its general vicinity.

Cause Identification: With a representation that is localized to where the bug might be, the specific cause of the bug can be pinpointed by further narrowing down shown representations and/or displaying levels of abstraction of the debugger state.

Solution Implementation: To validate the correctness of their solution for the bug, users can either continue to use their previous CD or update it to better expose the effects of their solution if the previous definition was not enough.

Consider an example of a bug where a *Player* loses one of their properties without making a trade. The user would like to see the behavior of the program by stepping through it, but the unconverted representation of the debugger state for each step would look like Figure 2.1, which is cluttered with irrelevant information. To better localize the bug, the user omits representations from parts they know for certain are not related to it, such as *DeckCards*, which are either Community Chest and Chance cards, or the arrays listing rent prices of *Properties*. The CD for these purposes is as seen in Listing 2.3 and an example of its resulting representation can be seen in Figure 2.5.

With the more focused view of Figure 2.5, the user discovers that when a *Player* loses a property without trading, its ownership is actually transferred to another *Player* that just landed on the square of the property. Apparently, the latter could purchase the *Property* of the *Square* which would usually only be possible if it was unowned. To identify the specific cause of this bug, the user can further refine the conversion to help identify its precise cause. Since the ownership of *Properties* is still correctly given to *Players* after purchase, the representations of unowned *Properties* do not need to be shown. Narrowing down the representation to only the *Players* and their owned *Properties* will allow the user to see exactly when the flaw occurs while

```
omit c:DeckCard;
omit c:DeckCard.f:effect;
omit c:BailCard;
omit c:DeckCardType;

omit c:MonopolyGame;
omit c:MonopolyGame.f:squares;

omit c:Property.f:rents;
```

LISTING 2.3: CD to help localize the property loss bug.

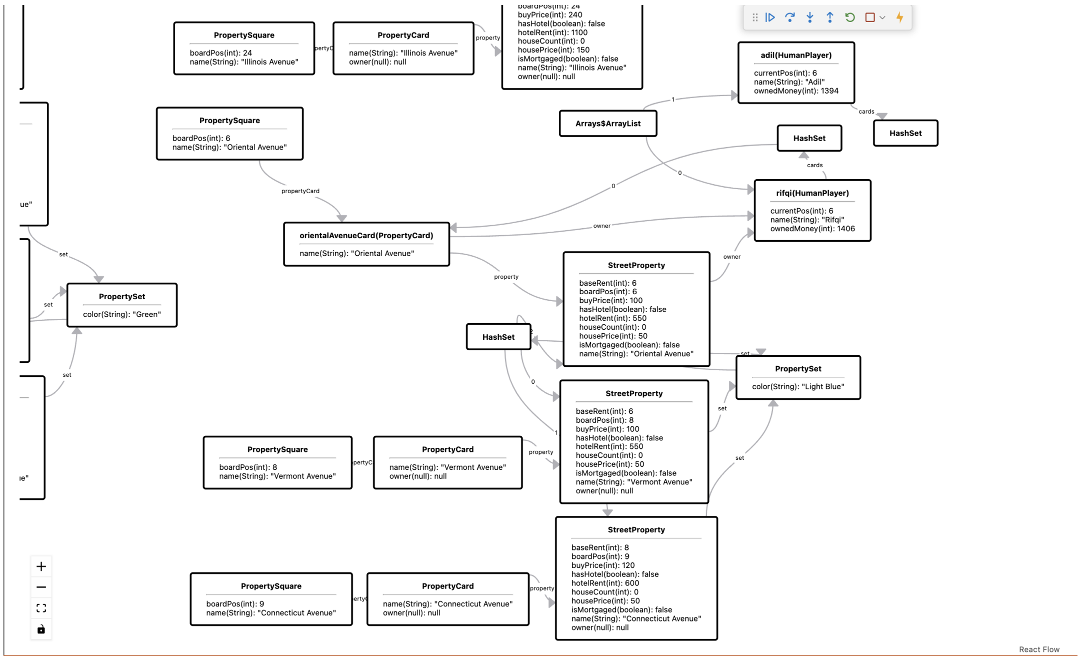
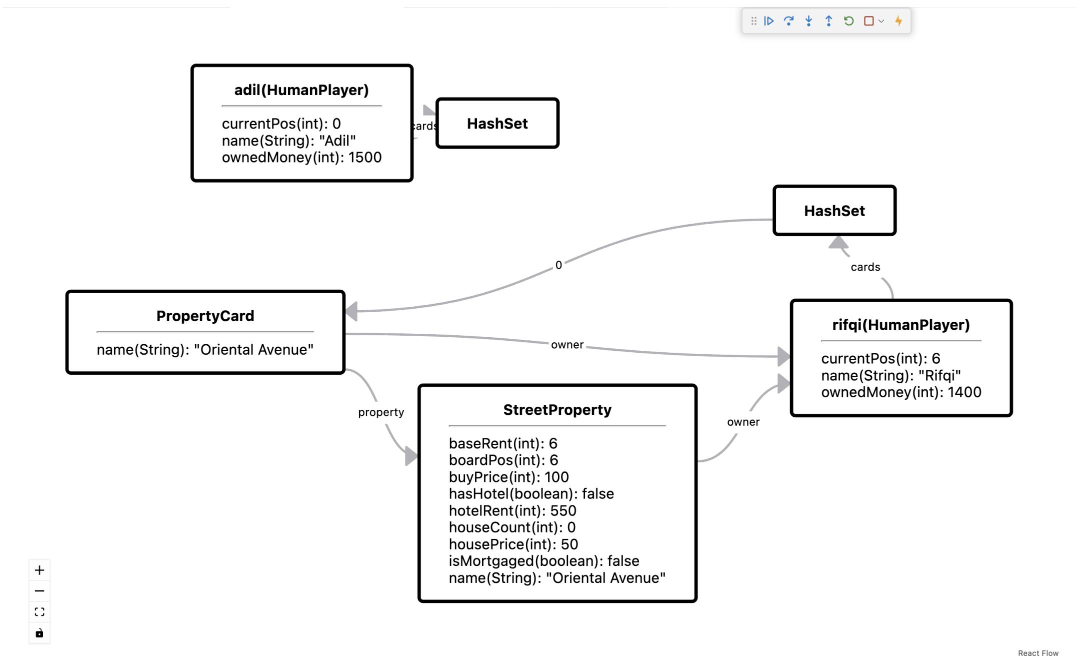


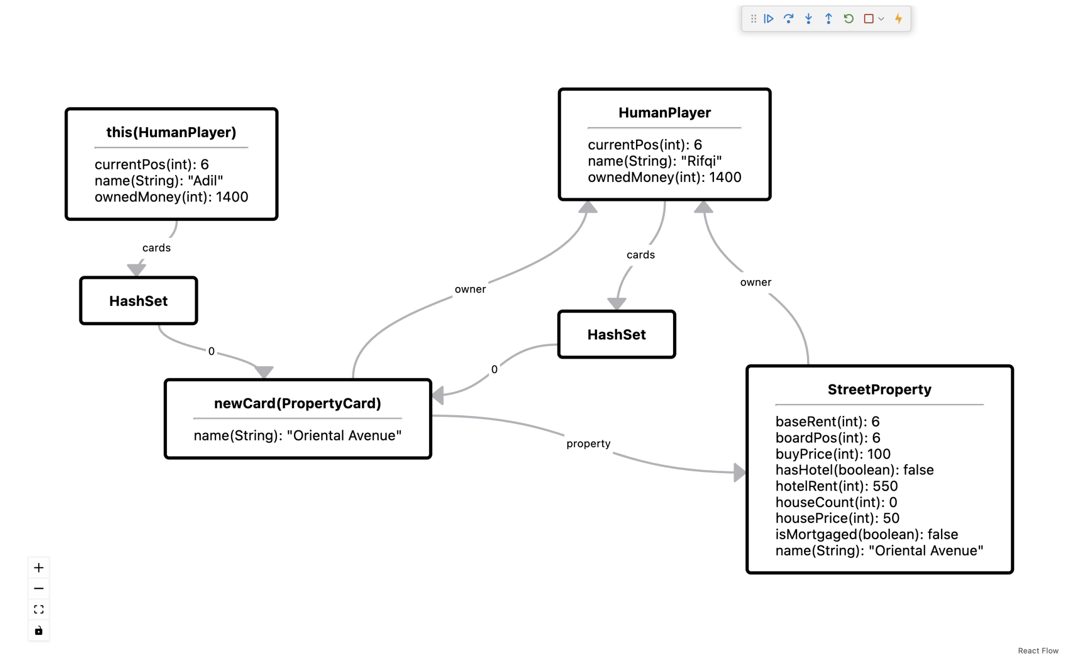
FIGURE 2.5: Converted representation of the Monopoly program to help localize the property loss bug.

stepping through the program, which is when a property is offered for purchase for the second time. Using the same CD as Listing 2.1, the view shown in Figure 2.6 is shown after ownership has been passed from the first owner to the next *Player* to land on the same *Square*.

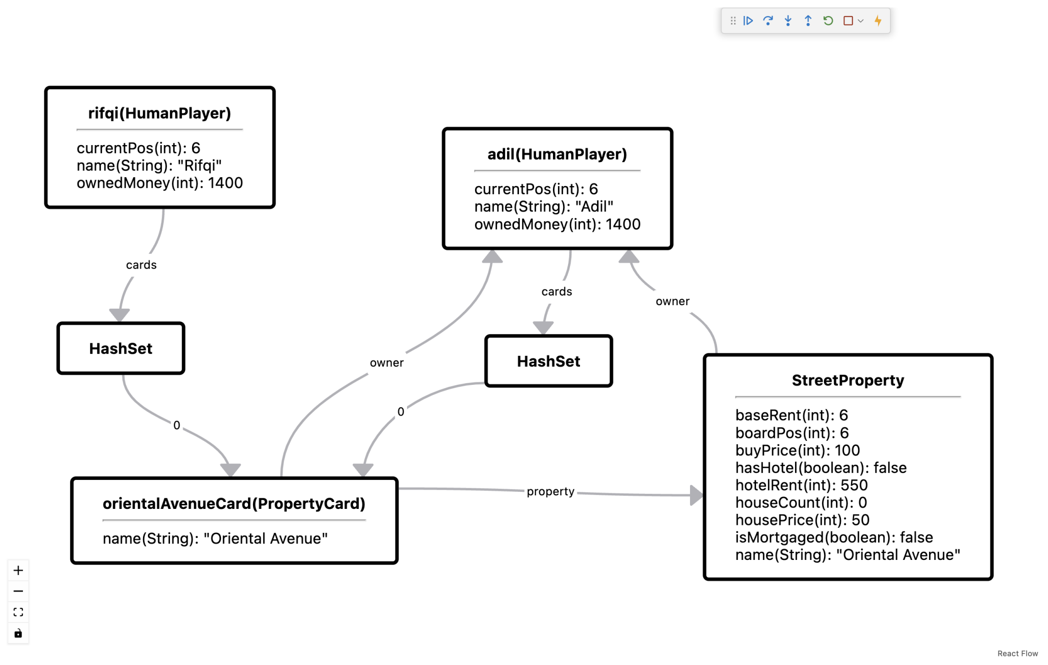
By stepping through the relevant parts of the code while examining changes made to the debugger state representation as shown in Figure 2.6, the user investigates the exact cause of the bug. Figure 2.6a shows the expected state after the first *Player* lands on the *Property* and buys it. Figure 2.6b shows the state after the second *Player* lands on that same *Property*. The user here can see that the card associated with the *Property* is an element of two different sets of cards each *Player* owns, which should not be possible. Note that this same information would not be very clear when shown in a TBD. Finally, Figure 2.6c shows that the *Property* is now owned by the second *Player*. The user finds that these two operations are done because no check is made to see if the *Property* of a *Square* is already owned and will be offered to whoever lands on it despite already being owned. This is confirmed by how both *Players* have exactly 1400 money each, which means 100 has been deducted from both of them, which is the price of the *Property* they bought.



(A) First player lands on the property.



(B) Card of property owned by the second player as well.



(C) Ownership of the property changes to the second player.

FIGURE 2.6: Snapshots of the representation while stepping through the code for cause identification.

The graphical representation changing between steps means that it is not static in time in what is called *Dynamic Graph Visualization* (DGV) [Bach, 2016, Beck et al., 2017]. Proper implementation to support this ought to significantly facilitate users in seeing differences in the representation at different steps. Note that the prototype used for the snapshots in Figure 2.6 has not yet implemented this correctly. For example, Figure 2.6b and 2.6c may not look too different at a glance but the position of the two *Players* nodes have been swapped due to the transfer in the *Property*'s ownership. Proper support for DGV would have clearly shown these differences.

After fixing the bug by adding a check of ownership of a *Property* upon landing on one, the user further adds to the CD to confirm the correctness of this solution. Specifically, the user will change the titles of *Player* nodes to their names and the money they have to ease reading relevant information from them. Listing 2.4 is an additional definition to Listing 2.1 that conveys this conversion and Figure 2.7 is the resulting representation.

```
// ...
c:Player {
    (nodeOf here).setTitle((valueOf f:name) +
        ": " + (valueOf f:ownedMoney));
}
```

LISTING 2.4: Additional CD to the one used for cause identification.

Figure 2.7 shows that after both *Players* have landed on the same *Square*, the owner of the *Property* has 1406 money while the other has 1494. This implies that the owner spent 100 money to buy the *Property*, and their opponent paid the 6 money rent to its owner upon landing on it after it has been purchased. This confirms that the solution to the bug is correct.

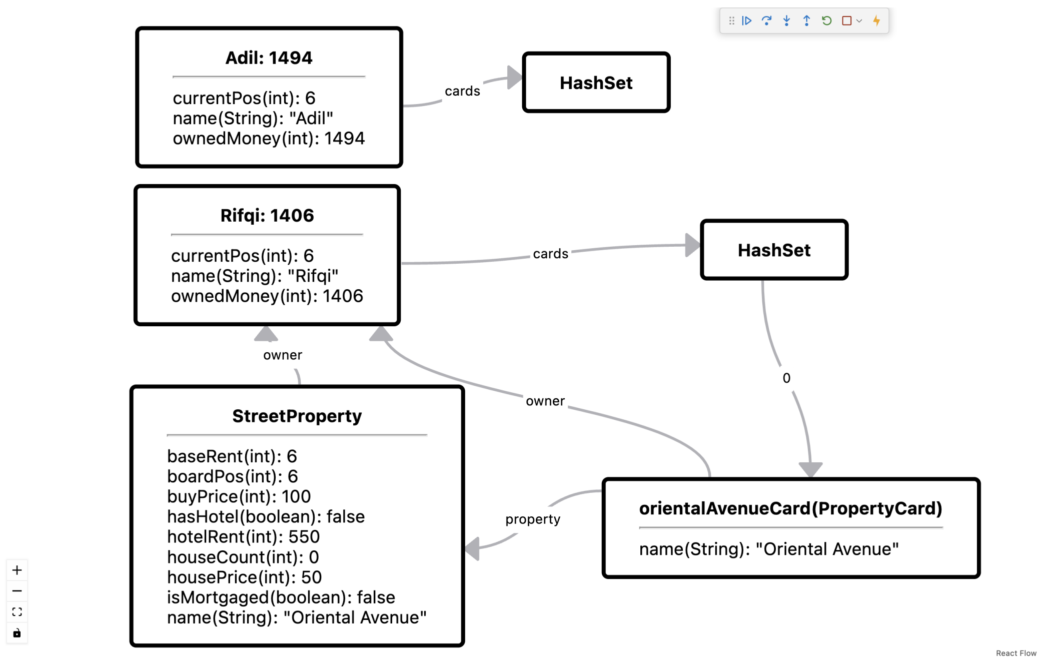


FIGURE 2.7: The converted representation to verify the solution.

Solving bugs in this way would most likely be incremental in that CDs can be written over time. For this, a feature that allows CDs to be reprocessed based on their changes during the debugging process without having to restart the debugger and the visualizer would save a substantial amount of time in the debugging process.

2.3 DSL Design

An external DSL is used for the current iteration of the prototype. Although the concept of UDRC is meant to be language-agnostic for imperative programming languages, the DSL described in this section was designed with class-based object-oriented programming languages in mind; where everything, including the main method, needs to be in a class, as in Java or C#. However, most of these design principles would still apply to other object-oriented imperative languages with only a few differences from their unique features. Although there may be a way to design the DSL in a way that covers all imperative languages, the time constraints of this study limit the capabilities of the design.

This section uses Listing 2.5 as a sample CD of the design principles discussed. For each *StreetProperty* of the debuggee:

1. If the `owner` field is null, omit the *StreetProperty* node from the representation (Line 3).

2. Otherwise, omit the edge from the *Property* to the owner and make a new one the other way around, instead (Lines 6-7).

3. Omit the node of the array of rent prices the *Property* has (Line 11).

The full grammar of the language has been written in Appendix A.

```
 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 newNode ownerNode thisNode;
 8         thisNode.setTitle(valueOf f:name);
 9     }
10
11     f:houseRents { omit nodeOf here; }
12 }
```

LISTING 2.5: CD to demonstrate DSL design choices.

2.3.1 Imperativeness

The language follows an imperative design where statements are sequentially executed, much like writing a new sequential program. This program written using the DSL is executed every time the debugger halts, which is when the debugger state is updated and the representation needs to be recalibrated and rerendered. Given this imperativeness, the statement types in the DSL include those that are commonly present in most imperative languages, such as variable assignment, if-else statements, and while/for loops. But given that it is a DSL with the primary focus on representation conversion, features not relevant to it are also excluded, such as concurrency and polymorphism, to remove unnecessary complexities.

The imperative design was chosen for the DSL for two reasons:

Familiarity: Users of the targeted imperative languages ought to already be familiar with the concept and mechanisms of the imperative design. Using a similar design for the DSL should minimize the effort users would need to spend to use the language.

Control: The DSL must provide users with detailed control such that users can define the comprehensive behaviors of their conversions. This is important as users would want specific conversions in a space of virtually infinite possible conversions. With imperativeness, users can better identify and specify CDs using intuitive sequential logic.

Listing 2.5 shows this characteristic. Lines 2 - 11 are sequentially executed for each *StreetProperty* in the debuggee. It also looks like a small program in itself.

2.3.2 Subjects

The DSL has a metaprogramming feature which is a new type called *Subject*. A *Subject* refers to an object in the debuggee. Its role is as a token to retrieve metaprogramming information about other objects related to the referred runtime object and to retrieve data about its representation. As *Subjects* represent runtime objects in the debuggee, they can also be chained in a way that mirrors property chaining in most languages.

In Listing 2.5, `here` is a *Subject* keyword which refers to the object of the current encompassing *location* (more explanation in Section 2.3.4). On Line 2, `here` refers to the *StreetProperty* whose representation is currently being converted. On that same line, the `here` *Subject* expression is being used to retrieve the node representing the *StreetProperty*. Similarly, on Line 11, `here` refers to the array object in the `houseRents` field of the current *StreetProperty*.

On Line 3, `f:owner` refers to the object that is the `owner` reference of the current *StreetProperty*. The expression in the condition checks if the current *StreetProperty* has the value `null` in the `owner` field, essentially checking if the property is owned. Additionally, the `owner`'s node is retrieved on Line 5 and is used for the conversions on Lines 6 and 7. Finally, Line 8 uses the *Subject* of the property's `name` reference to get the value of the reference in the form of a string and set it as its node's title.

2.3.3 Nodes, Edges, `show`, and `omit`

The two main components of the diagram subject to conversion are nodes and edges. Without no conversion applied, the diagram displays all nodes and edges, reflecting the current state of the program one-to-one. However, new nodes and edges can be created using the `newNode` and `newEdge` keywords, respectively.

Additionally, nodes and edges themselves have representation elements that they encapsulate. Each node has a title and any number of rows. Edges have optional labels. These sub-elements can be retrieved and changed through their encapsulating elements.

The two main command keywords for conversion of nodes and edges are `show` `omit`. When an element is *omitted*, it is removed from the representation but not deleted. This means that the object of the element still persists. Though, it can still be accessed and be made to show. When an element is *shown*, it is displayed in the diagram. If an element is both shown and omitted, the last command called on the element is applied in the rendered representation.

The command and expression keywords relating to representations and their conversions are explained in detail in Table 2.1.

Line 3 in Listing 2.5 shows the `omit` command being used to omit the node of *StreetProperties* if they are unowned. Line 6 omits all edges from the *StreetProperty* node to the node of its owner and Line 7 shows a new node in the opposite direction. Finally, the command in Line 8 changes the title of the *StreetProperty* node to the value of the property's `name` string.

2.3.4 Location

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. 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 and their semantics are as seen in Table 2.2.

In Listing 2.5, the scope opened in Line 1 is a class location for the *StreetPrpperty* class of the debuggee. Lines 2 - 11 are repeated for each *StreetProperty*. Therefore, each iteration is associated with a different *StreetProperty*, and the metaprogramming context of the iteration corresponds to the associated object. If among all *StreetProperties* are two — where *a* has an owner and *b* does not — then Lines 5 - 8 will be executed in the iteration of *a* while only Line 3 is executed in the iteration of *b*.

TABLE 2.1: Conversion commands and expressions and their semantics.

Keyword	Parameters	Returns	Explanation
`show`	`Node | Node[] | Edge | Edge[] | Subject | Subject[]`	-	Shows the given conversion objects or the nodes of the given *Subjects*.
`omit`	`Node | Node[] | Edge | Edge[] | Subject | Subject[]`	-	Omits the given conversion objects or the nodes of the given *Subjects*.
`nodeOf`	`Subject`	`Node`	Gets the node associated to the *Subjects*.
`nodesOf`	`Subject[]`	`Node[]`	Gets the nodes associated to the *Subjects*.
`edgesOf`	`Subject | Node, Subject | Node`	`Edge[]`	Gets the edges between the source and target nodes or those of the given *Subjects*.
`newNode`	`string`	`Node`	Creates a new initially omitted node with the given title.
`newEdge`	`Node | Subject, Node | Subject, string?`	`Edge`	Creates a new initially omitted edge with an optional label between the source and target nodes or the nodes of the given *Subjects*.

TABLE 2.2: Location types and their semantics.

Location Type	Block Semantic	Subject Semantic
Class `c`	Encompassed statements are applied for all objects of class `c`. Only valid as a top-most location.	An array of all debuggee objects that are instances of class `c`. Has no usage restriction.
Field `f`	Encompassed statements are applied for all `f` references of objects of the enclosing class or field block. Can be used in class or field location blocks.	The debuggee object that is the `f` reference of the debuggee object of the enclosing location block. Can be used in class or field location blocks.
Method `m`	Encompassed statements are executed when the runtime is currently halted inside method `m` of the class of the enclosing class location. Only valid directly in class locations.	Undefined
Local Variable `l`	Encompassed statements are applied for the local variable `l`. Valid as top-most location or in method locations.	The debuggee object with the same variable name in the current scope. Has no usage restriction.

Line 11 contains a field location for the `houseRents` field of the *StreetProperty* class. Inside the scope of the field location, the metaprogramming context is in the context of whatever object is the `houseRents` reference of the *StreetProperty* in the current iteration of the enclosing `StreetProperty` location. Consequently, the `here` keyword in Line 10 refers not to the *StreetProperty*, but rather to its array in its `houseRents` field.

Chapter 3

Prototype Implementation

A prototype for the concept of UDRC has been implemented as a Visual Studio Code (VSCode extension called *JIGSAW*[1] with Java as the target language for the programs subject to be debugged. Although this concept is language-agnostic for imperative programming languages as Chapter 2 and Section 2.3 mention, limitations in time only allow the prototype to work for only one target language.

3.1 The Prototype

Using JIGSAW, users would need to create a file called `spec.jig` at the top level of the project as a file where CDs can be written in.

To use JIGSAW while debugging, users would need to open the *JIGSAW View* window. Though writing a definition is optional, if the written definition includes errors during compilation, the view will not open and the error message will be shown. This ensures that the view has a valid CD that it follows, even if it is an empty CD.

After the view is opened, running the debugger will show the debugger state representation converted by the CD once the debugger hits a breakpoint and halts. In this view, nodes can be moved around by the user when needed. Continuing the execution of the debuggee until the next halt — such as stepping in, stepping out, stepping over, or resuming the program until another breakpoint is hit — causes the representation in the view to be rerendered based on the updated debugger state.

While JIGSAW and the debugger are still running, users can also update the CD and trigger JIGSAW to rerender the representation based on its updated version. Therefore, users do not need to restart JIGSAW and rerun the debugger every time changes are made to the CD, reducing the time needed to write them.

However, if the updated CD has an error during its compilation following the user's trigger, the definition will not be accepted and the error message will be displayed. Therefore, given how a valid definition is also required when initially opening JIGSAW, JIGSAW views always have valid CDs that they use in the process of displaying debugger state representations.

3.2 The Implementation

The implementation of JIGSAW is split into two main parts: the back end and the front end. The overview of the architecture and implementation can be seen in Figure 3.1.

[1]https://github.com/adilrifqi/jigsaw

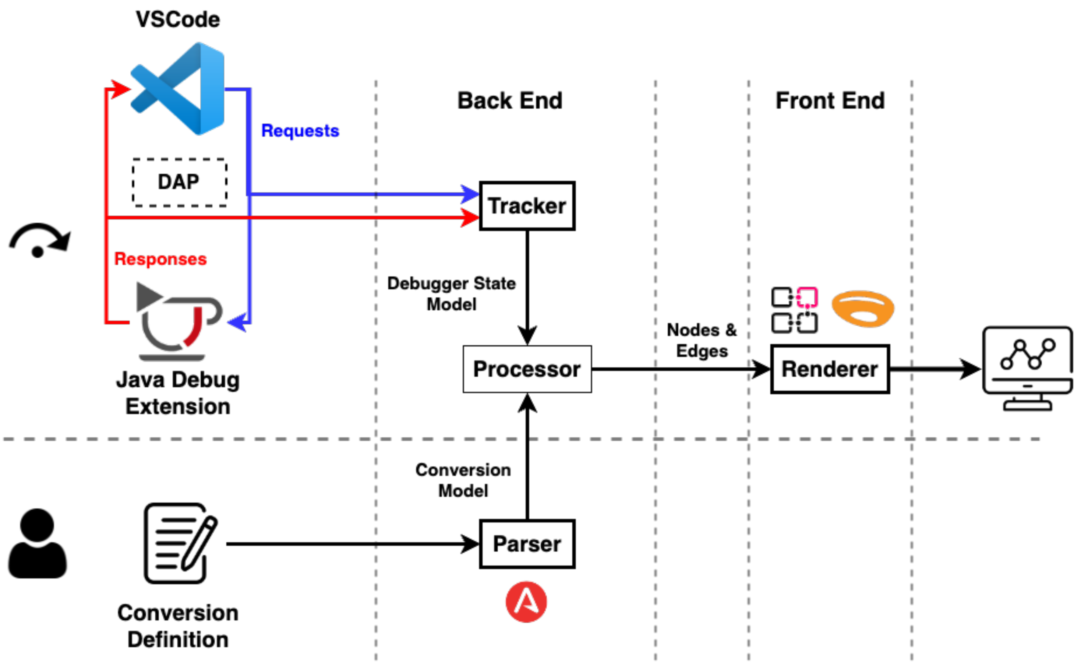


FIGURE 3.1: JIGSAW's Architecture.

3.2.1 The Back End

JIGSAW's back end also consists of retrieving two models: the debugger state model and the conversion model.

Debugger State Model Retrieval

Since the debugger state is updated every time the debugger halts — either by stepping in, stepping out, stepping over, or hitting a breakpoint in the execution of the debuggee — the back end needs to retrieve the updated debugger state at each halt.

VSCode has a protocol called the Debug Adapter Protocol (DAP)[2], where clients can be implemented for different programming languages that will communicate debug information to VSCode. Using this protocol, the VSCode debugger user interface (UI) can be used for all languages. Implementations of a language's DAP client are usually provided in the form of VSCode extensions. A DAP client for Java is provided in a VSCode extension called *Debugger for Java*[3].

At every halt, VSCode and the DAP client send each other messages as requests and responses. JIGSAW's back end uses VSCode's *DebugAdapterTracker*[4], to track their communications and eventually extract a debugger state model, which is then passed to the processor of the back end.

Conversion Model Retrieval

Unlike the debugger state model retrieval, retrieving the conversion model does not need to happen at every debugger halt. The main task of this part of the back end is to parse the CD written by the user and compile it into an internal model. Thus,

[2]https://microsoft.github.io/debug-adapter-protocol/
[3]https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-debug
[4]https://code.visualstudio.com/api/references/vscode-api#DebugAdapterTracker

this operation only happens on the user's command. This operation is part of the process of opening a JIGSAW view. It is also done when users trigger rerenders during debugging.

During this process, the string of the CD is passed to the parser generated using ANTLR[5]. The parser generates a parse tree which is then walked to generate the conversion model. Every time the conversion model is retrieved, it is passed to the processor of JIGSAW's back end.

Debugger State Model and Conversion Model Processing

Using the debugger state and conversion models passed on to it, JIGSAW's processor first generates the graph data of the nodes and edges that represent the debugger state. They represent the debugger state one-to-one.

The graph data is then processed further using the conversion model, essentially executing the CD "program" the user has written. The result from this process is then passed on to JIGSAW's front end.

3.2.2 The Front End

The front end receives the graph data from the back end. This is then used to generate the actual graph that will be rendered to the view, giving its nodes and edges dimensions.

The nodes of the graph are then laid out using elkjs[6] by providing it the generated graph. This gives each node a position that does not overlap with other nodes using an algorithm that makes the layout of the graph look organized.

Finally, React Flow[7] is used to render the graph and provide tools for navigating through and interacting with the debugger state visualizer.

[5]https://www.antlr.org/
[6]https://github.com/kieler/elkjs
[7]https://reactflow.dev/

Chapter 4

Case Study

For tools like this, user tests are evidently required to prove the effectiveness of the proposed feature. In doing user tests for this feature, however, the target metric would be the duration in which users debug programs because that is the intended goal of the feature's addition. Furthermore, this duration also includes the time it takes to understand the program, understand the effect of the bug, and the entirety of the debugging process. Even when these stages are compartmentalized to reduce variance in the measurements, numerous variables would still add noise to them.

To verify that the proposed feature makes a difference, a controlled experiment is required such that there is a control group that the experiment group is compared to. As such, two groups need to be determined. One consideration is the difficulty in balancing the two groups in terms of levels of expertise. As the tool is not particularly aimed toward novices, which are easier to balance, the variability in this aspect is substantial. Furthermore, experienced programmers most likely have a domain that they are more proficient in, which adds variance in terms of the time it would take to understand the buggy program participants are tested with. And finally, as the proposed feature aims to address size and complexity, not only would these large and complex programs need to be prepared, but it also adds complexity to how participants are to take the test. This final variable also exacerbates the first two as they may also affect the ways these programs are understood by participants.

Due to limitations in time and the complexity of doing user tests for this type of tool, this study provides fictional case studies to illustrate where, when, and for who the concept of UDRC is beneficial or not. This case study is also meant to illustrate the need for further research and real-world testing to validate the effectiveness and usability of the proposed concept.

The cases chosen in this case study involve personas with different expertise levels, backgrounds, and scenarios, which draws more robust and generalizable conclusions and avoids bias. Each case includes the decision-making process of the persona and a discussion of the conclusion that can be drawn from the case.

4.1 Novice Programmer Nancy

4.1.1 Scenario

Nancy is a first-year undergraduate Computer Science student. She is currently studying programming and debugging and is struggling to understand the concept of reference semantics, and thus is also having difficulty understanding the unexpected behavior of her code. She uses a debugger state visualizer to help herself understand which also happens to have user-definable representation conversion features.

```
 1 public class Sorter {
 2     private int[] arr;
 3     public Sorter(int[] arr) {this.arr = arr;}
 4
 5     public void sort() {Arrays.sort(this.arr);}
 6     public int[] getArray() {return this.arr;}
 7
 8     public static void main(String[] args) {
 9         int[] initialArray = {7, 4, 2};
10
11         Sorter sorter = new Sorter(initialArray);
12         sorter.sort();
13
14         System.out.println("Initial array: "
15             + Arrays.toString(initialArray));
16         System.out.println("Sorted array: "
17             + Arrays.toString(sorter.getArray()));
18     }
19 }
```

LISTING 4.1: Nancy's simple sorter program.

Nancy is working on a simple Java program written in Listing 4.1. She does not understand why the string printed by Line 14 includes the sorted array instead of the initial unsorted array, even though she never sorted the `initialArray` variable.

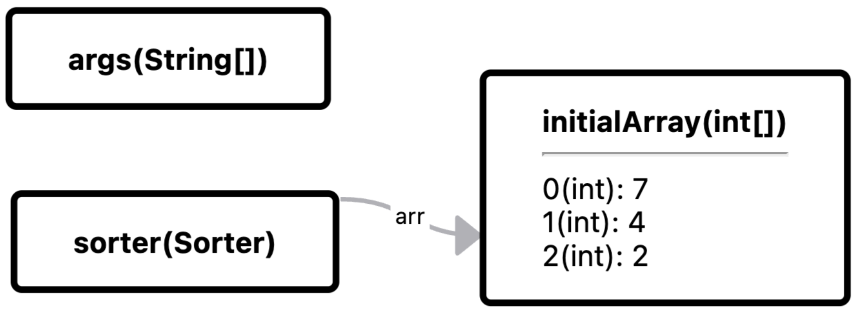


FIGURE 4.1: The representation of Nancy's sorter program when halted at line 12 of Listing 4.1.

Nancy uses the visualizer to better understand changes made to the debugger state while stepping through the program. Due to her program's small size and simplicity, she has no trouble seeing the entire state as in Figure 4.1 and the changes that happen. She discovers that the array in `sorter` has in its `arr` field points to the same array as what `initialArray` stores. She sees that after stepping over Line 5, the array both variables refer to are sorted. She concludes that both print statements in Lines 14 - 17 are printing the same array object.

She continues to write larger programs and continues to use the visualizer to help her understand her programs' behaviors, but she feels like there is now more information visualized at once. She recalls that the visualizer has a feature that helps with large and complex programs. But as the feature involves a DSL, which requires time and effort to understand on top of the language she is currently using the visualizer to learn, she stops using the visualizer for this program.

Nancy was able to understand Java mechanisms, her own code, and programming concepts better with the help of debugger state visualizations. She thinks that the representations in graph form are more accurate and explicit than what TBDs can show. She feels that it is not clear in TBDs when two variables refer to the same object. But she was intimidated by the complexity of the representation conversion features and the time she would need to invest in learning how to use them.

Nancy's feedback for the tool is that even though she finds the visualization feature helpful, she feels like the representation conversion feature could be simplified. Because she somewhat understands Java, she wishes to use Java or some other similar language instead of a separate external DSL for representation conversion. She also wishes to alternatively be able to use this feature interactively.

4.1.2 Discussion

This case demonstrates how novice programmers are non-target when it comes to using the user-definable representation conversion features, most prominently due to the DSL used to control them. Though she may only need to simply omit parts of her debugger state representations, it is likely intimidating to see a different language whose mechanisms she would need to understand to understand another language she is currently learning. A possible solution might be to change the DSL to one that does not require as much time and effort to learn and use, either by simplifying it or by utilizing knowledge its users already have.

4.2 Technical Lead Tina

4.2.1 Scenario

Tina is an experienced software developer and the team lead of her software development team. They are currently working on a project and need to ensure good code quality and collaboration between team members. She is interested in using the visualizer to aid in code reviews and communication.

Tina's team is currently working on an application that involves user members, their friends, and friend groups. Members (another word for users) can befriend other members. Members can also join groups, and this would automatically mutually befriend everyone else in that group. This means that any member of a group is friends with any other member of that group.

Before the meeting, Tina briefly checks the code of the preliminary *Member* and *FriendGroup* models. She suspects incorrectness in the model and would like to discuss it with the team. The preliminary model can be seen in Listing 4.2. Unfortunately, the data structure is not visualized well due to the abundance of nodes and edges representing friend relations between members, as seen in Figure 4.2. However, she was at least able to confirm her suspicion that Dave and Eve are not friends with each other despite being in the same *FriendGroup*.

As a very experienced programmer, she quickly became familiar with the representation conversion features and how to use them. She finds the documentation to

```java
class Member {
    String name; Set<Member> friends;
    public Member(String name) {
        this.name = name;
        this.friends = new HashSet<>();
    }
    public void addFriends(Member... newFriends) {
        this.friends.addAll(Arrays.asList(newFriends));
    }
}

public class FriendGroup {
    String name; Set<Member> members;
    public FriendGroup(String name) {
        this.name = name;
        this.members = new HashSet<>();
    }

    public void addMembers(Member... newMembers) {
        for (Member newMember : newMembers) {
            for (Member existingMember : members) {
                existingMember.addFriends(newMember);
                newMember.addFriends(existingMember);
            }
        }

        members.addAll(Arrays.asList(newMembers));
    }

    public static void main(String[] args) {
        Member alice = new Member("Alice");
        Member bob = new Member("Bob");
        Member charlie = new Member("Charlie");

        alice.addFriends(bob, charlie);
        bob.addFriends(alice, charlie);
        charlie.addFriends(alice, bob);
        FriendGroup group = new FriendGroup("");
        group.addMembers(alice, bob, charlie);

        Member dave = new Member("Dave");
        Member eve = new Member("Eve");

        group.addMembers(dave, eve);
    }
}
```

LISTING 4.2: Tina's team's preliminary model.

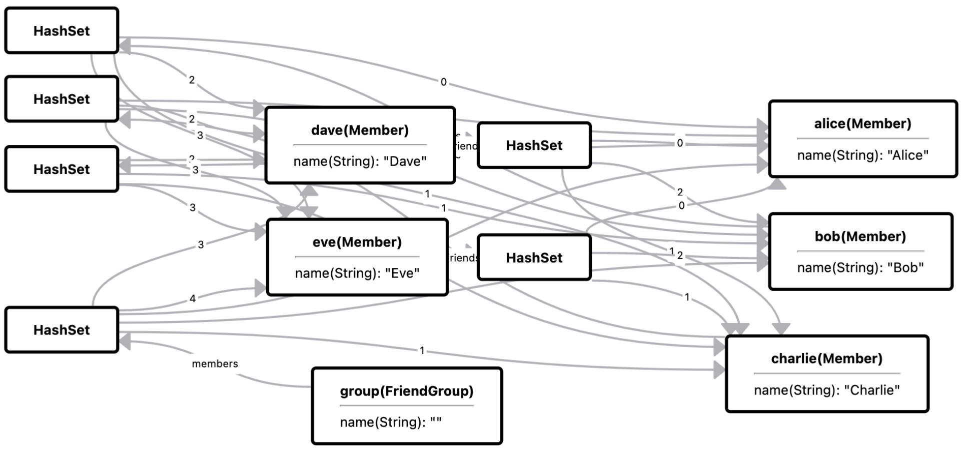


FIGURE 4.2: The representation of Tina's team's preliminary model.

```
c:Member {
    omit f:friends;
    for (Subject friend : childrenOf f:friends)
        show newEdge here friend;
}

c:FriendGroup {
    omit here; omit f:members;
    for (Subject member : childrenOf f:members)
        (nodeOf member).addRow
            ("Member of " + valueOf f:name);
}
```

LISTING 4.3: Tina's CD.

be straightforward due to the familiarity she has with the imperative constructs in the DSL and was able to grasp the representation conversion and metaprogramming features of the DSL rather quickly due to her experience. For the code review, she prepares a CD to help her discuss the code with her team. The CD she wrote can be seen in Listing 4.3 and the converted representation of the same debugger state shown in Figure 4.2 can be seen in Figure 4.3.

During the meeting, the converted representation helped Tina show that members Dave and Eve are not friends with each other despite being in the same group. By manually stepping through the code line-by-line they discover that the `addMembers` method of the *FriendGroup* class mutually befriends each new member of the group with each existing member of the group, but not to each other. Meaningful and concise discussions were possible due to the elimination of distractions which may consist of visual clutter or having to separately illustrate models and behaviors in the meeting. Additionally, Tina also shared the CD in Listing 4.3 with her team members for them to use while they do their own work.

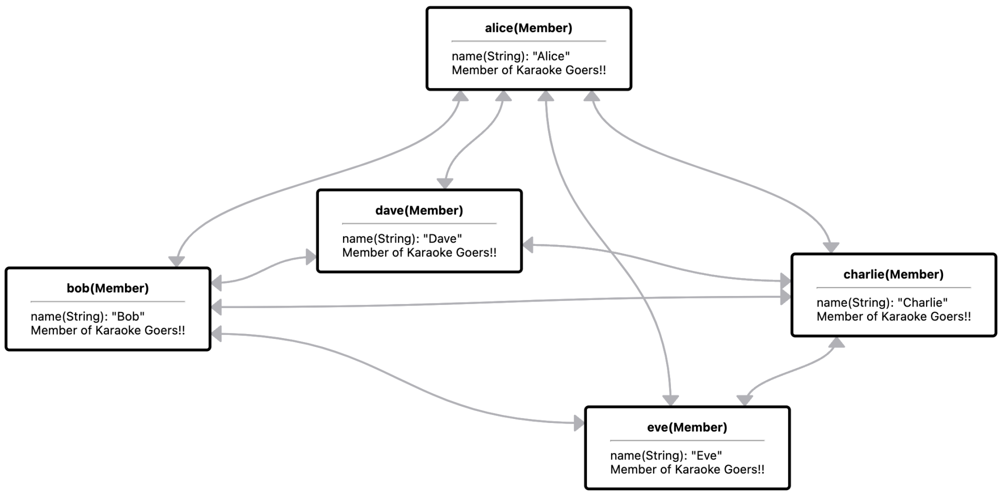


FIGURE 4.3: The representation of Tina's team's preliminary model using the CD in Listing 4.3.

4.2.2 Discussion

This case demonstrates how the tool can be leveraged to facilitate collaboration and communication among people with sufficient programming experience and knowledge. In this case, Tina was able to manage a concise discussion as her conversion allowed her team members to understand the debugger state more easily and what it means in terms of the program's behavior and flaws. This is due in large part to the focus representation conversion provides.

Additionally, this case also highlights that the representation conversion feature along with its current DSL design targets programmers with sufficient experience. Here, Tina was able to grasp the concept of the DSL relatively quickly due to her experience and expertise.

Furthermore, this case also emphasizes the advantages of both visualizations and representation conversion. Only using a TBD for Tina's code review would have made it cumbersome to draw friend relationships on a whiteboard for the discussion, especially if they would need to redraw it at every halt of the debugger. But even by using graphical representations such as the one shown in Figure 4.2, reading the debugger state while stepping through the program would not be very convenient for the meeting.

Finally, this case also shows one significant advantage of CDs, which is that they can be shared with other people, such as coworkers, such that the time and effort spent on writing them is minimized as others would only need to reuse the shared CDs.

4.3 Multi-Language Programmer Max

4.3.1 Scenario

Max is a programmer who is enthusiastic about knowing about and working with different programming languages. Max's primary goal is to optimize his productivity as a multi-language programmer. He works on different projects that involve a mix of languages, and their debugging requirements can vary significantly based on the specific language and the complexity of the codebase. Max is motivated to find a tool that can seamlessly support debugging across different languages and provide consistent and convertible representations of debugger states.

To better understand the differences if programming languages, he often practices by creating pet projects using languages he wishes to learn. He is currently working on two different projects, each built using a different programming language. Currently, his projects are a game called "Galactic Adventures" written in Java, and a text auto-complete tool called "TextAutoComplete" written in Python.

Java Project - Galactic Adventures

"Galactic Adventures" is an action-packed 2D space shooter game developed using Java. Players control a spaceship and navigate through various challenging levels, battling alien invaders and asteroids.

Max finds a bug where projectiles meant for enemies are also affecting the player's spaceship, and vice versa. This leads to inaccurate targeting when using power-ups that automatically target supposed enemies, unintended damage to players and enemies, and more.

Max initially suspects that there might be issues with team assignments. He thinks that units, including the player's and enemies' spaceships and bullets fired from them, might be assigned to incorrect teams, thus causing incorrect behavior when colliding with other units. He investigates using the visualizer first without any conversion to the representation as seen in Figure 4.4. For the purposes of this case, as the entirety of the program can neither be explained nor written here, representations of its simplified version will be shown instead.

Fortunately, as Max has had extensive experience in learning new programming languages in the past, he rapidly understands the concept of the DSL the representation conversion feature of the visualizer requires, especially since it mostly uses common language constructs. He creates a CD that omits nodes and edges irrelevant to him to see the issue. The converted representation can be seen in Figure 4.5.

He finds that the problem was that all ships, including the player's and enemies', use the same bullet pool. This means all bullets fired by the player and enemy are taken from the same pool. Each team was supposed to have different pools and each pool was supposed to have a uniform bullet behavior. Max eventually fixes this bug.

Python Project - TextAutoComplete

"TextAutoComplete" is a Python-based text auto-completion tool. It helps users quickly find and suggest word completions based on the input provided. As with text prediction applications, this application heavily uses the trie data structure.

Max would find, however, that his application suggests a string that is neither a valid word nor a valid term. He investigates using the same visualizer he used to debug his Galactic Adventures game. Using his experience with using the tool, he was able to quickly write a CD that he can use to help him understand the behavior of his program without having to spend more time determining how to use a new visualizer for a different language. He was able to quickly write his CD as seen in Listing 4.4 to convert the representation as shown in Figure 4.6 to the one in Figure 4.7.

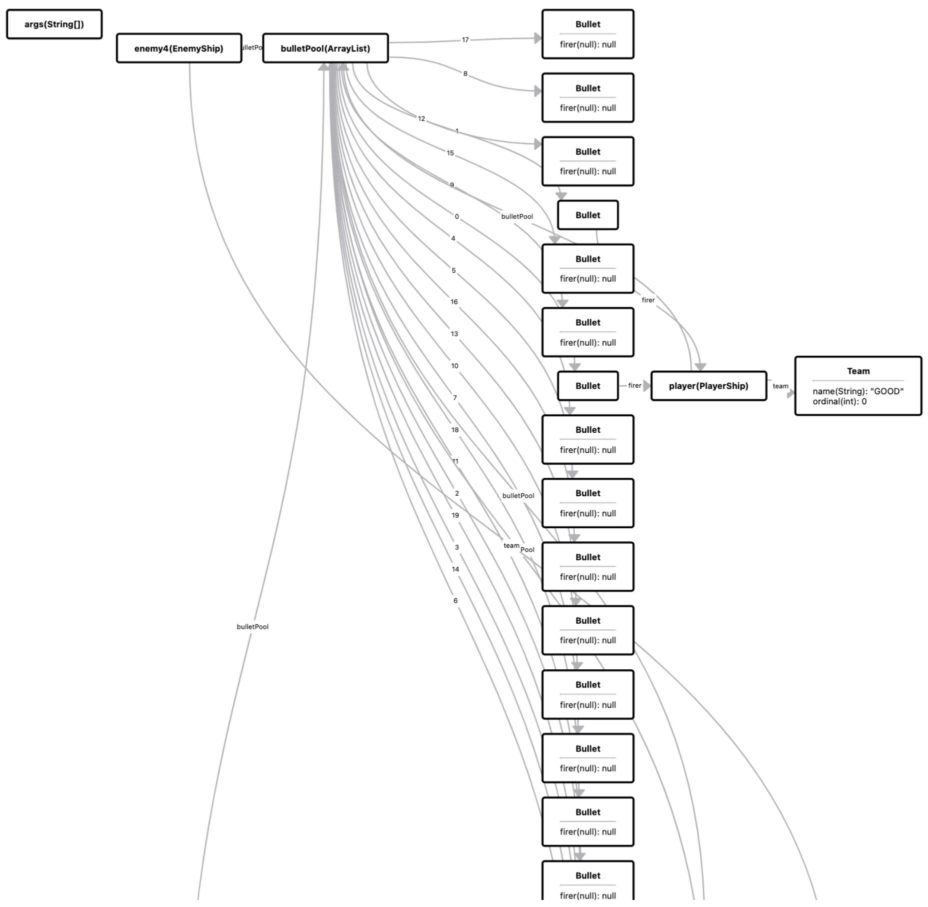


FIGURE 4.4: The representation of Max's simplified Galactic Adventures program.

Even though Max added *ie* as a word, it is not suggested when *i* is typed. Instead, *ied*, *ien*, *iea*, and *iench* are suggested. Figure 4.7 shows why this is the case. By halting the execution for each word added to the trie while seeing changes made to its representation, Max can investigate the incorrect behavior done to the trie. Figure 4.7a shows the initial state of the trie, only containing its root node. Figure 4.7b shows how the trie changes after a word is added with Figure 4.7c displaying how the trie looks like after a few more words have been added without any incorrect behavior yet observed. Differences between Figures 4.7c and 4.7d show the bug in how new words are added. Apparently, while adding the word *ie* to the trie in the former figure, it reuses a node wrapping the letter *e* that is already the child of another that wraps the letter *t* instead of creating a new node for the letter, giving former node an additional parent. Something similar happens in the addition of the word *teach* with the *c* node.

4.3.2 Discussion

This case shows how beneficial the language-agnostic property of the design is. Due to the unavoidability of having to learn how to use new tools, language-agnosticism allows users to reduce the effort they have to exert as they can use the same visualizer again for projects in different languages. After learning how to control the

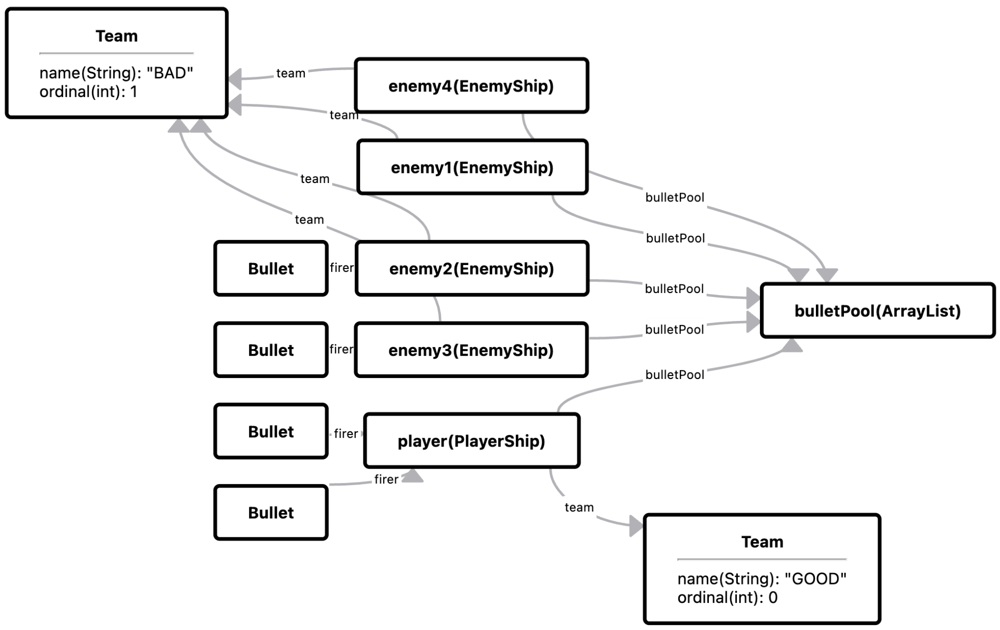


FIGURE 4.5: The converted representation of Max's simplified Galactic Adventures program.

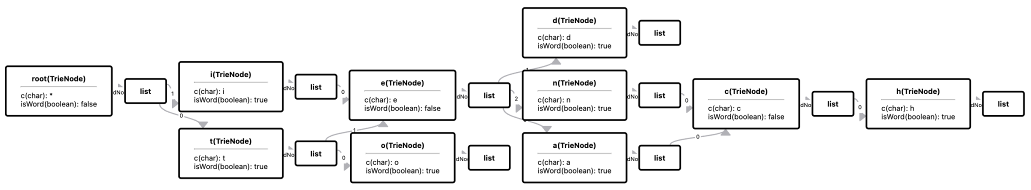
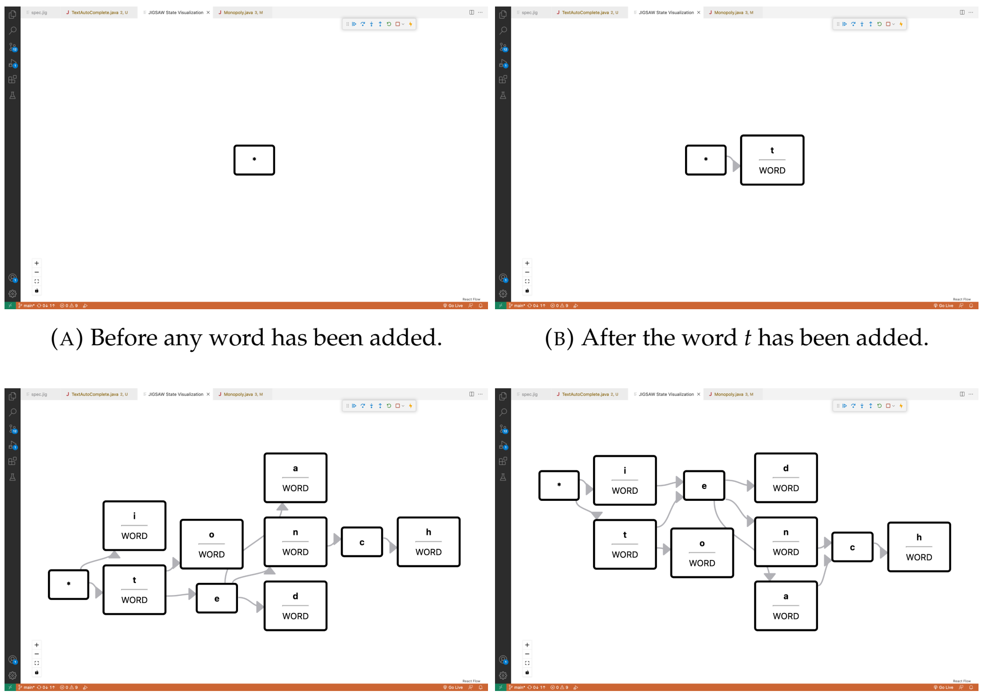


FIGURE 4.6: The representation of Max's simplified TextAutoComplete program.

representation conversion feature once while debugging his Galactic Adventures program, Max does not need to do it again for other projects written in other programming languages.

This case also shows the versatility of the visualizer. It can be used to help understand the behavior and identify bugs in programs of different natures and designs. On top of the visualizer displaying debugger states as intuitive nodes and edges, this versatility is also due in large part to the design choice of using a DSL for user control. It allows users to convert the representations of their program's debugger states unconstrained to the visualizer's options and capabilities if it were a GUI-based tool.

However, Max finds that the visualization lacks proper support for DGV, making it not very intuitive and easy to see relevant differences in the representations between steps. In this case, he notices that the representation swaps the position of the *d* and *a* nodes between the steps shown in Figures 4.7c and 4.7d. He wishes that nodes remain the same as much as possible so that changes on them are as clear



(A) Before any word has been added. (B) After the word *t* has been added.

(C) After more words have been added without incorrect behavior. (D) After the words *ie* and *teach* have been added, showing incorrect behavior.

FIGURE 4.7: The converted representation of Max's simplified TextAutoComplete program.

as possible. In this case, the position of the nodes ought to not change. Though this did not affect his bug's cause identification, he thinks that this would obstruct the behavior comprehension process if it happened to have swapped the relevant nodes.

4.4 Data Scientist Sarah

4.4.1 Scenario

Sarah is an experienced data scientist with a passion for exploring and analyzing large datasets to derive valuable insights and make data-driven decisions. She works on various projects involving data cleaning, feature engineering, model development, and statistical analysis. As a data scientist, Sarah frequently uses Python and specialized libraries like Pandas, NumPy, and scikit-learn to perform her data analysis tasks.

Sarah is currently working on a predictive maintenance project for a manufacturing company. The goal is to develop a machine-learning model that predicts equipment failures and maintenance requirements based on sensor data collected from the manufacturing process. The project involves preprocessing large volumes of sensor data, engineering relevant features, building predictive models, and evaluating model performance.

However, she notices that the model's accuracy varies significantly across different folds of the cross-validation process. Sarah is also familiar with the domain of predictive maintenance and she notices discrepancies in the model's predictions compared to her knowledge and what she would expect.

```
c:TrieNode {
    omit f:childNodes;
    for (Subject childTrieNode : childrenOf f:childNodes)
        show newEdge here childTrieNode;

    Node thisNode = nodeOf here;
    bool isWord = valueOf f:isWord;

    thisNode.clearRows();
    if (isWord) thisNode.addRow("WORD");
    thisNode.setTitle(valueOf f:c);
}
```

LISTING 4.4: Max's CD for his TextAutoComplete program.

Sarah decides to use the debugger state visualizer in the hopes that the representation conversion feature can narrow down her focus to the relevant data and expose the culprit of this behavior. Sarah first debugs her program without defining a conversion to see what she will be working with. The representation of a debugger state can be seen in Figure 4.8.

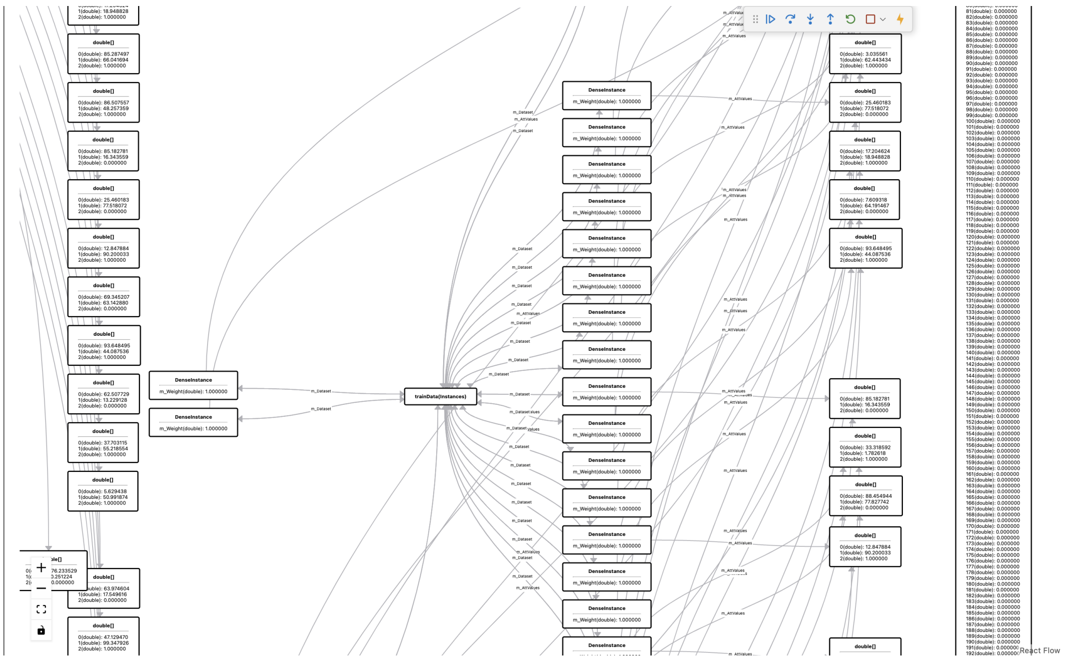


FIGURE 4.8: The representation of Sarah's Predictive Maintenance program.

From Figure 4.8, it is clear that debugger states would mostly contain numerical values. She finds the representation of objects and references to be insignificant because her main concern is on these numerical values and changes made to them and because her program doesn't involve many references that she writes herself. Nonetheless, Sarah tries to convert this the way debugger states are represented to attempt to comprehend changes made to the model while stepping through her program. However, she finds it difficult to display these values intuitively using nodes and edges no matter what conversion she comes up with.

Normally, she would display these values using scatter plots or line plots such that she can compare the original target variable values against the modified values for each data point in the training data. These visualizations can provide a picture of patterns or discrepancies that might indicate the presence of added noise or incorrect operations.

As she sees no way to do this using the visualizer, she decides to stop trying to use the visualizer to solve the bug in her program and use domain-specific tools and libraries tailored to data science tasks instead to effectively debug, analyze, and gain insights from her predictive maintenance project in manufacturing.

4.4.2 Discussion

Although the debugging tool with user-definable representation conversion proves effective for traditional software development and general programming tasks, Sarah, as a data scientist, finds that it does not align well with her specific data science needs. The tool's graphical representation of debugger states as nodes and edges is not directly suited for the complexities of data analysis, model development, and statistical evaluation, which are critical components of data science projects.

In other words, this case highlights that no matter what conversion is done on graphical representations of debugger states, the representation would still be in the form of a graph consisting of nodes and edges. Its strong point is displaying object relations step-by-step to expose behavioral discrepancies to the user. It most often works well on programs whose states can be represented relationally, especially commonly used data structures. But when the behavior and state of the program are not optimally representable using nodes and edges, there is little representation conversion can do. Consequently, this problem also applies to other domains, such as natural language processing, image and video processing, etc.

Chapter 5

Related Work

5.1 UML Object Diagrams in Program Comprehension

Torchiano conducted an experiment [Torchiano, 2004] 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 [Torchiano et al., 2017] 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.

Bach [Bach, 2016], Beck et al. [Beck et al., 2017] discuss DGV as the visualizing graphs that can change over time. The former presented *GraphDiaries* which implements techniques to aid DGV. Animations are used to smoothly transition between separate time intervals, all the while maintaining the entirety of the screen for both the graph and its arrangement. Thumbnails are also used which possess inherent compactness, which enables the display of multiple time intervals concurrently. The latter study proposed a taxonomy to classify DGV techniques based on different factors, including visualization approach, interaction techniques, and representation strategies. The paper discusses the challenges and trends in DGV and provides insights into the strengths and limitations of different approaches. The visualizer UDRC is integrated to still uses graphs for state representations, as with other visualizers, meaning it must show the representations and their changes over time. DGV techniques are evidently aligned with the purposes of UDRC in the temporal department and may be used in conjunction.

5.2 Scalability Features in Visual Debuggers

Java Interactive Visualization Environment (JIVE) [Lessa, Czyz, and Jayaraman, 2010] offers interactive features to display how a Java program runs at various levels of detail. Among its features is a debugger state visualizer in the form of an object diagram. JIVE's solutions to managing large executions are:

Exclusion Filters: Exclude representations of objects from packages, libraries, or classes.

Code Interval Debugging: Only visualize representations of debugger states in an interval of the program. Useful for the focused debugging of specific packages or subsystems without visualizing the entire program execution.

Dynamic Slicing: Changes to the selected object or variable will be isolated to focus on a particular object of interest.

Though they may somewhat resolve visual clutter, it does not have the granularity and control UDRC provides. With the proposed language, these solutions can also be defined along with other conversions the user might need.

BlueJ [Bennedsen and Schulte, 2010] 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 [Cazorla and Viejo, 2015] 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.

5.3 Customization of Representations

Velázquez-Iturbide [Velázquez-Iturbide and Presa-Vázquez, 1999] proposed the integration of visualizations customization into WinHIPE [Pareja-Flores, Urquiza-Fuentes, and Velázquez-Iturbide, 2007], which is the Windows version of a programming environment for the functional programming language Hope [Burstall, MacQueen, and Sannella, 1980]. This feature allows the programmer to customize the visualization of intermediate expressions resulting during any evaluation. Customizations users can choose are:

Text vs. Graphics: Intermediate expressions can be shown using a textual or binary tree representation, or a mix of both.

Typographic Styles: Change how texts look; including their font, font style, font size, and color coding.

Visualization Simplification: Visualizations are distorted to show the most relevant parts and hide the less important ones using a technique called "fisheye views".

This was a step closer to faster program comprehension and better visualization readability. UDRC has a similar goal for imperative languages where visualizations are of the debugger state. The proposed representation conversion is essentially *customizing* what and how information is displayed, only on a more detailed and larger scale.

Chapter 6

Future Work

6.1 DSL Improvement

Although the design of the language allows users to control the behavior of their conversions with great detail, it is still far from ideal.

Although a substantial portion of the language's statements and expressions resemble those from regular general-purpose imperative languages, the most important constructs of the language are those that are not commonly found in them. This mostly includes the location constructs and metaprogramming features. Users would need to invest a significant amount of time to learn how the language is used. Furthermore, the language has its own set of syntaxes, for which users would need to constantly consult the language's documentation.

Moreover, users would also be required to devote effort beyond what should be necessary to specify the behavior of their conversions. This is apparent from the listings that have previously been shown. For example, Listing 2.2 requires 16 lines of code, and Listing 2.5 requires 12 lines of code for their relatively trivial conversions. Needless to say, more specific or complex conversions, especially those that considerably convert the abstraction of representations, would require more time and effort to define.

Ultimately, the problems of the language can be summarized as its learning and usage overhead. The rest of this section discusses current solution concepts to these problems.

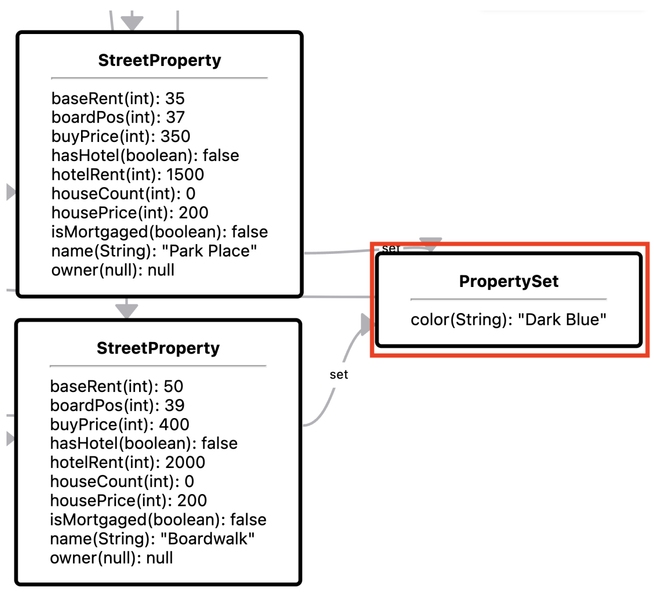
6.1.1 Shortcuts

An intuitive solution to the effort overhead in writing CDs is to provide *shortcuts* to reduce them. Shortcuts are function-like constructs in the DSL that perform a sequence of predefined conversions. The purpose of these constructs is to trivialize commonly written conversions or those that will be written multiple times in different parts of the definition.

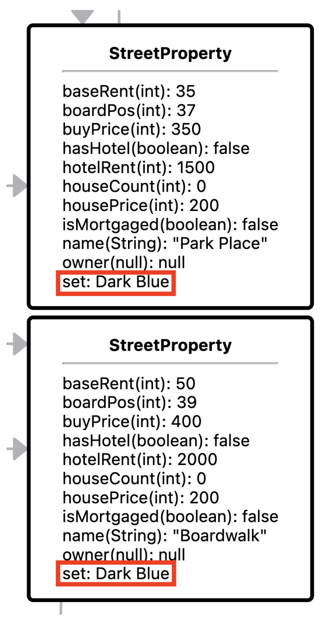
An example of a possible shortcut would be one that *inlines* references to objects. In other words, this shortcut would do the following:

- Omit the nodes of the targets.
- For each target, a new row would be added to the nodes of objects that reference the target containing its `toString()` representation.

Figure 6.1 shows the usage of this shortcut. Figure 6.1a shows a part of the Monopoly program's unconverted representation, where each property has a reference to a *PropertySet* (boxed in red), which is represented by a color. The information of each *PropertySet* can be inlined to the nodes of objects that reference it. By including `inline c:PropertySet;` in the CD, the same part of the same debugger state can



(A) Without `inline` on `PropertySet`.



(B) With `inline` on `PropertySet`.

FIGURE 6.1: Plain and inlined `PropertySet`.

instead be represented as in Figure 6.1b. Here, the rows boxed in red are the new rows that inline the information of each property's `set` reference.

Shortcuts can also do as much as completely change the representation of data structures, such as representing lists as binary trees. Instinctively, this ought to relieve the costs of writing in the language somewhat. To reinforce the concept of a construct resembling a function, users should be able to define their own shortcuts in addition to the preexisting ones. However, incorporating this feature could considerably complicate the process of learning the language. Luckily, there may still be a potential resolution to this issue.

6.1.2 Internal DSL

Transitioning to an internal DSL within a general-purpose programming language should help reduce the learning curve associated with the language, particularly when the host language and the debugger's target language are one and the same. Additionally, it would also lower the effort required for writing definitions since users would have access to the host language's existing capabilities and the support provided by IDEs for that language.

Initially, the decision to provide an external DSL was based on the intention to create a focused language with only essential features. However, subsequent iterations in the language's development revealed that it increasingly resembled a general-purpose language, especially after the inclusion of custom shortcuts. Moreover, one of the primary purposes of an external DSL is to serve as an intermediary language between domain experts and developers, which, in this scenario, refers to the same group of individuals.

6.1.3 Implicit Statements

The present structure of the DSL intentionally distinguishes the metaprogramming capabilities from the aspects related to converting representations. This distinction is particularly noticeable in how subjects are distinct from their representation properties. This is in large part due to subjects being tokens used to retrieve both metaprogramming values and representation elements. Consequently, users are compelled to compose distinct statements for establishing context and performing conversion tasks. Using Listing 2.5 as an example, the user needs two separate explicit expressions to check the nullity of `f:owner` in Line 3 and to retrieve its node representation in Line 5.

By erasing this division and establishing implicit associations between the target program and its representation, the act of defining conversions can be simplified and made more intuitive. Consider the following preliminary illustration: assuming there is a *PropertyCard* named *pc* that contains a reference to a *Property* named *p*, writing `pc.p.title = pc.p.name;` in the CD would result in updating the node title of *p* to match the value of its `name` string.

6.2 UI/UX Improvement

The UI of the prototype currently only allows users to do basic interactions, such as zooming and moving nodes around. The experience in using the visualizer may be significantly improved with more enhanced functionalities. This section discusses improvement ideas that can be made to the front end.

6.2.1 Dynamic Graph Visualization

Currently, only static snapshots of the state's representation are shown every time the debugger halts without much indication of the changes made between halts. Users need aid in DGV as seeing changes in between halts is part of the effectiveness of these visualizers.

Nodes must only be moved as little as possible between halts such that users can observe their changes. Furthermore, animations and thumbnails may also be helpful in between halts to further expose these changes, as [Bach, 2016] describes. In addition to this, it ought to also be useful to be able to view the state representations of previous halts. Finally, visual cues that highlight changes, such as coloring, node or row highlighting, which most conventional TBDs also employ, should be intuitive additions for this purpose.

6.2.2 Text Search Feature

One example of a useful feature would be the ability to search nodes or edges using text. This would further help users in navigating the view in addition to the help their conversions have provided for them. Like most search functionalities, moving between found items ought to pan the camera to those items. This would especially be helpful if the user is interested in finding representations with unique names of identifiers.

6.2.3 Interactive Conversion

Currently, the only way to make changes to how debugger states are represented is by changing the CD, no matter how small that change may be. *Interactive conversion* consists of two parts: interactive representation changes and automatic definition editing.

Interactively changing debugger state representations entails that users can make basic changes to the representation by interacting directly with the GUI. For example, users would be able to omit a node by right-clicking on it and selecting `omit`. This functionality ought to also include the option to apply the change to other instances of the same representation. For example, omitting a node of an instance of a class would also provide the option to omit all other nodes of that class's instances. Though, the details of how this would work for node or edge addition are still unclear.

A complementary feature to this would be to have changes made interactively in the GUI to automatically be written in the CD. For example, interactively omitting all nodes of instances of the *Property* class would automatically add `omit c:Property;` to the definition. But the details of how this would work have also not been investigated. Although, it is likely that this functionality would only mostly work on simple conversions. Conversions that require more logic, like those that require calculations, conditions, or loops, probably would still need to be defined manually.

With the addition of these features, users would be able to both interact with the representation and also define its behavior through code.

6.3 User Tests

User tests in the form of a controlled experiment would eventually need to be conducted to demonstrate the usefulness of representation conversion. Solutions to the variables mentioned in Chapter 4 would first need to be thoroughly investigated to conduct such an experiment.

Chapter 7

Conclusion

Using the graph structure to represent debugger states mitigates the problems of TBDs and also provides the benefits visualization brings. However, this solution itself brings forth the problem of visual clutter and abstraction gap in large or complex programs, effectively nullifying the solution using the graph structure provides.

To solve these two problems, the feature *user-definable representation conversion* is proposed for visualizers that use graphs as debugger state representations for users to convert state representations. Depending on their circumstances, users can convert these representations to choose which and how information of the debugger state is displayed, such that they can focus on it as they step through the program. The feature is foreseen to be useful for users during different steps of the debugging process. For this purpose, user control and versatility are essential in defining the behavior of conversions, for which a DSL is provided.

A major challenge of this work is in the design of this DSL as it should require the minimum learning and usage overhead to make this feature a viable option in debugging. The DSL is imperative to provide familiarity and control. It also has contextual features from which users can extract metaprogramming and representation contexts.

A prototype for this concept has been developed as a VSCode extension called JIGSAW, which currently only works for Java. JIGSAW tracks messages sent between VSCode and Java's debugger extension to create a debugger state model and compiles the user's CD to generate the graph data to be displayed and passes it to the front end to be rendered. JIGSAW has basic UI features such as zooming and moving nodes but does not yet allow for interactive graph manipulation. CDs updated in the middle of a debug session can be used without restarting the debugger or visualizer.

For this study, a fictional case study is developed to consider how programmers of different levels of expertise and fields interact and perceive the proposed feature. Beginners may find visualization useful but may find it difficult to use representation conversion features. However, experienced programmers working on multiple programming languages and projects may appreciate the language-agnosticism and versatility of the feature. Furthermore, representation conversion is thought to be useful for collaboration and communication. Although, representation conversion for graphical representations may not be very helpful for domains that typically do not consider object relations the primary part of their debugging focus, like data science, for instance.

However, there are considerable tasks that still lie ahead for this study. Numerous improvements can be implemented in both the representation conversion concept and its prototype to reduce the required resources in learning and using the feature. Furthermore, user tests are still necessary to concretely evaluate the effectiveness of this feature in meeting its targeted purpose.

Bibliography

Alhumaidan, Fahad and Nazir Ahmad Zafar (2014). “Possible improvements in UML behavior diagrams”. In: *2014 International Conference on Computational Science and Computational Intelligence*. Vol. 2. IEEE, pp. 173–178. DOI: 10.1109/CSCI.2014.113.

Bach, Benjamin (2016). “Unfolding dynamic networks for visual exploration”. In: *IEEE Computer Graphics and Applications* 36.2, pp. 74–82. DOI: 10.1109/MCG.2016.32.

Beck, Fabian et al. (2017). “A taxonomy and survey of dynamic graph visualization”. In: *Computer graphics forum*. Vol. 36. 1. Wiley Online Library, pp. 133–159. DOI: 10.1111/cgf.12791.

Bennedsen, Jens and Carsten Schulte (2010). “BlueJ visual debugger for learning the execution of object-oriented programs?” In: *ACM Transactions on Computing Education (TOCE)* 10.2, pp. 1–22. DOI: 10.1145/1789934.1789938.

Burstall, Rod M, 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*, pp. 136–143. DOI: 10.1145/800087.802799.

Cazorla, Miguel and Diego Viejo (2015). “JavaVis: An integrated computer vision library for teaching computer vision”. In: *Computer Applications in Engineering Education* 23.2, pp. 258–267. DOI: 10.1002/cae.21594.

Holy, Lukas et al. (2012). “Lowering visual clutter in large component diagrams”. In: *2012 16th International Conference on Information Visualisation*. IEEE, pp. 36–41. DOI: 10.1109/IV.2012.17.

Lessa, Demian, Jeffrey K Czyz, and Bharat Jayaraman (2010). “JIVE: A pedagogic tool for visualizing the execution of Java programs”. In: *Bericht, Univ. of New York, Buffalo*.

Mernik, Marjan, Jan Heering, and Anthony M Sloane (2005). “When and how to develop domain-specific languages”. In: *ACM computing surveys (CSUR)* 37.4, pp. 316–344. DOI: 10.1145/1118890.1118892.

Pareja-Flores, Cristóbal, Jamie Urquiza-Fuentes, and J Angel Velázquez-Iturbide (2007). “WinHIPE: An IDE for functional programming based on rewriting and visualization”. In: *ACM SIGPLAN Notices* 42.3, pp. 14–23. DOI: 10.1145/1273039.1273042.

Sadiku, Matthew et al. (2016). “Data visualization”. In: *International Journal of Engineering Research And Advanced Technology (IJERAT)* 2.12, pp. 11–16.

Strobelt, Hendrik et al. (2018). “S eq 2s eq-v is: A visual debugging tool for sequence-to-sequence models”. In: *IEEE transactions on visualization and computer graphics* 25.1, pp. 353–363. DOI: 10.1109/TVCG.2018.2865044.

Torchiano, Marco (2004). “Empirical assessment of UML static object diagrams”. In: *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, pp. 226–230. DOI: 10.1109/WPC.2004.1311064.

Torchiano, Marco et al. (2017). “Do UML object diagrams affect design comprehensibility? Results from a family of four controlled experiments”. In: *Journal of Visual Languages & Computing* 41, pp. 10–21. DOI: 10.1016/j.jvlc.2017.06.002.

Velázquez-Iturbide, JÁ 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. DOI: 10.1109/FIE.1999.841580.

Appendix A

DSL Grammar

⟨program⟩ ::= ⟨statement⟩*

⟨statement⟩ ::= ⟨location⟩ | ⟨command⟩

⟨location⟩ ::= ⟨loc-type⟩ ⟨ID⟩ (‘`.`’ ⟨loc-type⟩ ID)* ‘`{`’ ⟨statement⟩* ‘`}`’

⟨loc-type⟩ ::= (‘`c:`’ | ‘`f:`’ | ‘`m:`’ | ‘`l:`’)

⟨command⟩ ::= ⟨scope⟩ | ⟨if⟩ | ⟨while⟩ | ⟨for⟩ | ⟨semiless⟩

⟨scope⟩ ::= ‘`{`’ ⟨command⟩* ‘`}`’

⟨if⟩ ::= ‘`if`’ ⟨conditional⟩ (‘`else`’ ‘`if`’ ⟨conditional⟩)* (‘`else`’ ⟨command⟩)?

⟨while⟩ ::= ‘`while`’ ⟨conditional⟩

⟨for⟩ ::= ‘`for`’ ‘`(`’ ⟨semiless⟩? ‘`;`’ ⟨expr⟩? ‘`;`’ ⟨semiless⟩? ‘`)`’ ⟨command⟩ | ‘`for`’ ‘`(`’ ⟨type⟩ ⟨ID⟩ ‘`:`’ ⟨expr⟩ ‘`)`’ ⟨command⟩

⟨semiless⟩ ::= ⟨type⟩ ⟨ID⟩ ‘=’ ⟨expr⟩ | ⟨expr⟩ ‘=’ ⟨expr⟩ | ⟨conversion⟩ | ⟨expr⟩

⟨conditional⟩ ::= ‘`(`’ ⟨expr⟩ ‘`)`’ ⟨command⟩

⟨conversion⟩ ::= ‘`omitAll`’ | ‘`showAll`’ | (‘`show`’ | ‘`omit`’) ⟨expr⟩

⟨expr⟩ ::= ⟨disjunction⟩

⟨disjunction⟩ ::= ⟨conjunction⟩ (‘`||`’ ⟨conjunction⟩)*;

⟨conjunction⟩ ::= ⟨comparison⟩ (‘`&&`’ ⟨comparison⟩)*;

⟨comparison⟩ ::= ⟨sum⟩ (‘<’ | ‘<=’ | ‘==’ | ‘`!=`’ | ‘>=’ | ‘>’) ⟨sum⟩ | ⟨sum⟩

⟨sum⟩ ::= ⟨sum⟩ (‘+’ | ‘-’) ⟨term⟩ | ⟨term⟩

⟨term⟩ ::= ⟨term⟩ (‘`*`’ | ‘/’ | ‘`%`’) ⟨negation⟩ | ⟨negation⟩

⟨negation⟩ ::= (‘`-`’ | ‘`!`’)? ⟨suffixed⟩

⟨suffixed⟩ ::= ⟨suff-prop⟩ | ⟨suff-arr⟩ | ⟨suff-fields⟩ | ⟨primary⟩

⟨suff-prop⟩ ::= ⟨suffixed⟩ ‘`.`’ ⟨ID⟩ ‘`(`’ (⟨expr⟩ (‘`,`’ ⟨expr⟩)*)? ‘`)`’

⟨suff-arr⟩ ::= ⟨suffixed⟩ ‘`[`’ ⟨expr⟩ ‘`]`’

⟨*suff-fields*⟩ ::= ⟨*suffixed*⟩ ('.' 'f:' ⟨*ID*⟩)+

⟨*primary*⟩ ::= ⟨*ID*⟩ | ⟨*diag-elem*⟩ | ⟨*subject-rule*⟩ | ⟨*value-of*⟩ | ⟨*literal*⟩ | ⟨*par-expr*⟩ | ⟨*array-expr*⟩ | ⟨*parents-expr*⟩ | ⟨*is-null*⟩ | ⟨*new-map*⟩ | ⟨*inc*⟩

⟨*value-of*⟩ ::= 'valueOf' ⟨*expr*⟩

⟨*array-expr*⟩ ::= '[' (⟨*expr*⟩ (',' ⟨*expr*⟩)*)? ']'

⟨*par-expr*⟩ ::= '(' ⟨*expr*⟩ ')'

⟨*parents-expr*⟩ ::= ('parent' '.')+ ⟨*ID*⟩

⟨*is-null*⟩ ::= 'isNull' ⟨*expr*⟩

⟨*new-map*⟩ ::= 'newMap' '<' ⟨*type*⟩ ',' ⟨*type*⟩ '>'

⟨*diag-elem*⟩ ::= ⟨*new-node*⟩ | ⟨*new-edge*⟩ | ⟨*node-of*⟩ | ⟨*nodes-of*⟩ | ⟨*edges-of*⟩

⟨*new-node*⟩ ::= 'newNode' ⟨*expr*⟩

⟨*new-edge*⟩ ::= 'newEdge' ⟨*expr*⟩ ⟨*expr*⟩ ⟨*expr*⟩?

⟨*node-of*⟩ ::= 'nodeOf' ⟨*expr*⟩

⟨*nodes-of*⟩ ::= 'nodesOf' ⟨*expr*⟩

⟨*edges-of*⟩ ::= 'edgesOf' ⟨*expr*⟩ ⟨*expr*⟩

⟨*subject-rule*⟩ ::= 'parents' | 'parentsOf' ⟨*expr*⟩ | 'here' | 'children' | 'childrenOf' ⟨*expr*⟩ | ('c:' | 'f:' | 'l:') ⟨*ID*⟩

⟨*inc*⟩ ::= ('++' | '-') ⟨*ID*⟩ | ⟨*ID*⟩ ('++' | '-')

⟨*literal*⟩ ::= ⟨*num-lit*⟩ | ⟨*string-lit*⟩ | ⟨*bool-lit*⟩

⟨*num-lit*⟩ ::= ((⟨*nonzero*⟩ ⟨*digit*⟩+) | ⟨*digit*⟩) ('.' ⟨*digit*⟩*)?

⟨*string-lit*⟩ ::= '"' ⟨*any-char*⟩* '"'

⟨*bool-lit*⟩ ::= 'true' | 'false'

⟨*type*⟩ ::= ⟨*basic-type*⟩ | ⟨*type*⟩ '[' ']' | 'Map' '<' ⟨*type*⟩ ',' ⟨*type*⟩ '>'

⟨*basic-type*⟩ ::= 'num' | 'bool' | 'string' | 'Node' | 'Edge' | 'Subject'

⟨*letter*⟩ ::= [a-zA-Z]

⟨*digit*⟩ ::= [0-9]

⟨*nonzero*⟩ ::= [1-9]

⟨*any-char*⟩ ::= `[a-zA-Z0-9!@#$%^&*()_+{}:"<>?,./;'[\]\\`~ -]`