What You See Is Not Always What You Get: An Analysis of Informative Graphs in Formal Methods Languages

By Lucia Regina Reyes Gonzalez

Computer Science Honors Thesis

Brown University, May 2021

Advisor

Tim Nelson

Brown University Computer Science

Reader

Shriram Krishnamurthi

Brown University Computer Science

Table of Contents

Abstract

Model-finding tools, such as the Alloy Analyzer and Forge, are used to model and examine systems through the use of mathematical formulas and expressions. After defining the basic structure of a system, along with its constraints and expected behavior, the user is able to either view solution instances that satisfy the constraints or verify desirable properties about their model. Many researchers have looked into enhancing these model-finding tools with various new user-interface extensions, one of them being Amalgam [6]. Amalgam is an extension to Alloy that shows the user, for every output instance, which modifications can be made without violating the model's constraints; Amalgam calls unchangeable portions of instances locally necessary. This tool is also capable of producing proof-based explanations, known as provenance, for these necessities [6].

This thesis analyzes the usefulness of Amalgam*'s* local necessity to increase a user's understanding of their model and aid the debugging process of identifying over-constraints and under-constraints in their code. In order to test whether local necessity was useful to users, I ran multiple user studies, including a study in the course CSCI 1710: Logic for Systems at Brown University and two rounds of studies in Amazon's Mechanical Turk. While the results from the study run in CSCI 1710 were not reliable enough to make an analysis, I found statistically significant results from the Mechanical Turk studies that suggest that some ways of displaying local necessity might be confusing to non-expert users. Additionally, I was unable to find a statistically significant advantage of using Amalgam's local necessity, potentially contradicting findings in Nelson et al. (2017). This thesis also describes the process of implementing Amalgam in Forge, a formal methods language used in CSCI 1710. In addition, I describe the benefits and disadvantages of implementing such a tool in Forge in order for this paper to act as a guide for future developers working on the language.

Keywords
Invariants, Formal Methods Languages, Model, Alloy, Amalgam, Locally Necessary, Local Necessity, UNSAT Cores, Provenances, Under-constraints, Over-constraints, Forge, Sterling.

1. Introduction

"My code is correct, but doesn't work" is a common phrase amongst programmers. Past research has demonstrated how one of the biggest challenges for programmers is identifying bugs in their own programs [1]. However, not enough emphasis has been placed on why programmers believe that their code is correct, even if it does not work. The main cause for this could be a lack of understanding of the problem at hand; if programmers were to fully understand a task before starting it, they would be more likely to identify what required invariants are not holding.

The same happens with students. Previous research has demonstrated that students begin programming assignments without having a complete understanding of the problem [2, 3, 4]. This raises the question of how can a student's understanding of a programming task improve before they start their implementation?

Formal methods tools are used to model and analyze systems through the use of mathematical formulas and expressions. *Alloy* [5] is such a tool where, initially, the user creates a *model*[1] by defining the basic structure of a system, along with its constraints and expected behavior. Afterwards, the user is able to either view solution instances that satisfy the constraints or verify desirable properties about their model (if verification fails, Alloy produces a counterexample instance). Many researchers have made improvements to Alloy, one of them being Amalgam [6]. *Amalgam* is an extension to Alloy that shows the user, for every output instance, which modifications can be made without violating the model's constraints; Amalgam calls unchangeable portions of instances *locally necessary*. This tool is also capable of producing proof-based explanations, known as *provenance*, for these necessities. In summary, Amalgam gives users access to two features: listing the local necessity of a given instance in the form of tuples, and for each tuple, getting its respective provenance(s) [6].

While the idea of local necessity makes mathematical sense, nobody has yet answered the question of whether it really helps to guide users through their models and allow them to understand their programs better. The field of formal methods languages is feared by many computer scientists, as they are "intimidated by the mere idea of formal specifications, which they fear may be too 'mathematical' for them to understand and use." [7] Would there be a way of diminishing this fear by utilizing Amalgam's local necessity?

[1] This thesis follows the notational standard of the Alloy community. We use 'instance' to refer to a relational structure that satisfies or doesn't satisfy the model which is a set of constraints over some relation or signature.

Throughout this thesis, I aim to answer the question of whether Amalgam's local necessity has the potential to increase a programmer's understanding of the task at hand. I also seek to determine whether local necessity is useful for programmers to understand their models in formal methods languages. In order to determine this, this thesis will present multiple user studies done both at Brown University's course *CSCI:1710 Logic for Systems* and *Mechanical Turk,* a popular crowdsourcing platform.

Additionally, I will provide an overview of my implementation of Amalgam in *Forge,* the main programming language used in CSCI 1710. This overview will also outline the benefits and drawbacks of implementing this tool in Forge with the purpose of acting as a guide for future developers working in that same language.

2. Motivation

2.1 Motivation for Topic

The main motivation for this thesis came from Nelson et al. (2017). This paper outlined the implementation of provenances and local necessity in Amalgam, and I was impressed by the tool's potential to help users trace over-constraints and detect under-constraints. A model is *under-constrained* if it is satisfied by unintended models and *over-constrained* if it does not satisfy intended models [8]. However, while reading Nelson et al. (2017) I noticed that they did not mention any type of user study to sustain the claim of Amalgam's usefulness. With this in mind, I tried the tool myself on a simple model and started thinking about possible ways to test its value through user studies.

Initially, I thought about testing the usefulness of both provenances and local necessity. However, after doing more research, I found that Danas et al. (2017) had already looked into the usefulness of Amalgam's provenances. In their study, Danas et al. (2017) compared the utility of UNSAT cores versus Amalgam's provenances in a debugging task for an upper-level course of computer science at Brown University [9]. However, this study did not look into the possibility of exposing students to local necessities. Given this area of opportunity, there was motivation to investigate the usefulness of local necessity to aid debugging processes as well as increase users' understanding of their programming task.

User Studies of Principled Model Finder Output: A study by Danas, Nelson, Harrison, Krishnamurthi & Dougherty

If a given model is unsatisfiable, *UNSAT cores* are a subset of the model that is in itself unsatisfiable; having this subset allows users to narrow down the area where the potential bug could be. Danas et al. (2017) presented students with a feline rendition of the "Connections of Kevin Bacon" game in a laboratory for the course CSCI 1710. The model, coded in Alloy, can be seen in figure 1.

Figure 1. KittyBacon's Connections Model with Highlighted UNSAT Core

```
/*01*/ sig Cat {friends : set Cat}
/*02*/ fact NoFriendlessCats {no c:Cat | no c.friends}
/*03*/ fact NoSelfFriendship {no c:Cat | c in c.friends}
/*04*/ fact SymmetricFriendship {friends = ~friends}
/*05*/
/*06*/ one sig KittyBacon extends Cat {}
/*07*/ fun F[c:Cat]:set Cat {c.friends}
/*08*/ fun FF[c:Cat]:set Cat {F[F[c]]-F[c]-c}
/*09*/ fun FFF[c:Cat]:set Cat {F[F[F[c]]]-F[F[c]]-F[c]-c}
/*10*/ fun connectionsOf[c:Cat]:set Cat {F[c]+FF[c]+FFF[c]}
/*11*/ pred Connected {Cat-KittyBacon=connectionsOf[KittyBacon]}
/*12*/
/*13*/ pred SConnected {Cat-KittyBacon in KittyBacon.^friends}
/*14*/ assert IsSuperConnected {Connected iff SConnected}
/*15*/ check IsSuperConnected for exactly 3 Cat
/*16*/ check IsSuperConnected for exactly 4 Cat
/*17*/ check IsSuperConnected for exactly 5 Cat
/*18*/
/*19*/ one sig CCC {members : set Cat }
/*20*/ fact CoolCatClub {CCC.members=connectionsOf[KittyBacon]}
/*21*/
/*22*/ // UNSAT CORE QUERY
/*23*/ pred KittyBaconIsCool {KittyBacon in CCC.members}
/*24*/ run KittyBaconIsCool for exactly 4 Cat
/*25*/
/*26*/ // PROV QUERY: why not CCC.members(CCC$0,KittyBacon$0)
/*27*/ run {} for exactly 4 Cat
```

(Danas et. al (2017))

Line 1 defines the `Cat` signature and lines 2-4 define the way that friendship between `Cats` works. `NoFriendlessCats` enforces all `Cats` having at least one friend, `NoSelfFriendship` enforces that no `Cat` can be friends with itself, and `SymmetricFriendship` enforces that if `CatA` is friends with `CatB`, then `CatB` must be friends with `CatA`. Line 6 is the definition of `KittyBacon` and lines 7-11 define the bounded transitive closure operator `ConnectionsOf[Cat]`. Lines 13-17 show a comparison between bounded and unbounded notions of transitive closure. Lines 19-20 define the `CCC` signature and restrict the `members` of `CCC` to be the connections of `KittyBacon`. Line 23 defines the predicate `KittyBaconIsCool`, which checks whether `KittyBacon` is part of the `members` of `CCC`. Line 24 runs this predicate.

Students in the study were asked to write a free-response answer to why the predicate `KittyBaconIsCool` failed, meaning that `KittyBacon` was not a part of the `CCC members`. In addition, they were asked to select a fix for the bug out of three provided edits. The free-response answer was coded into which predicates students "blamed" for the failure of the predicate; this could be any combination between the predicates `NoSelfFriendship`, `ConnectionsOf[Cat]`, and `CoolCatClub`. The three edits shown to students were (1) updating the `CCC`'s `members` definition to be the union of `ConnectionsOf[KittyBacon]` and `KittyBacon` (correct fix), (2) allowing self-friendship (violating the `NoSelfFriendship` predicate), and (3) adding `KittyBacon` to its own connections (invalidating the definition of the bounded transitive closure).

The 28 students working with UNSAT cores were shown the red and pink constraints highlighted in figure 1, which are responsible for unsatisfiability. The 35 students working with Amalgam's provenances were shown two provenances, both subsets of the UNSAT core. One of the provenances was the same as the UNSAT core minus the `NoSelfFriendship` highlight, and the second provenance was the same as the former minus `ConnectionsOf[Cat]`.

The results show that one-fifth of students that worked with provenance proposed an edit that invalidated the model's constraints. The results for students working with UNSAT cores and provenances can be seen in figures 2 and 3 (reproduced from Danas et al. (2017)).

Figure 2. Results for Students working with UNSAT cores

Constraint	# Student Blames	# Student Edits	Correct?
CoolCatClub	18 (64%)	22 (79%)	Y
ConnectionsOf[Cat]	27 (96%)	0 (0%)	N
NoSelfFriendship	14 (50%)	1 (4%)	N
No Edit	n/a	5 (18%)	N

Figure 3. Results for Students working with Provenances

Constraint	# Student Blames	# Student Edits	Correct?
CoolCatClub	20 (57%)	23 (66%)	Y
ConnectionsOf[Cat]	21 (60%)	6 (17%)	N
NoSelfFriendship	9 (26%)	0 (0%)	N
No Edit	n/a	6 (17%)	N

2.2 Motivation for User Studies

Since Danas et al. (2017) focused on Amalgam's provenances, I saw the opportunity to imitate their study utilizing local necessity instead. This would allow me to make a direct comparison between the usefulness of local necessity and provenance. Additionally, I was also inspired by previous research and experiences focused on increasing a student's understanding of an assignment before approaching it.

In the area of previous research, I was motivated by how Wrenn and Krishnamurthi tested whether writing input-output examples before starting an assignment could increase a student's understanding [10]. Drafting these examples has proven successful in the past to avoid errors in an assignment [11], and they were interested in testing whether students would continue writing these examples even if they were not required to do so. In order to do this study, they analyzed the adherence of writing examples in a semester-long introductory computer science course at Brown University. Wrenn and Krishnamurthi augmented students' programming environment to encourage them to develop examples before starting an assignment and designed a measure to assess whether students

continued writing examples throughout the course. Their study showed that students adhered to writing examples, contrary to previous research on the subject [10].

In the area of previous experiences, I thought back to my time as a student for CSCI 0150: Introduction to Object-Oriented Programming. In this course, students are asked to develop graphic programs including popular games such as Tetris and Doodle Jump. However, in order to guide students through the project and give them an idea of what the final outcome should look like, students are given access to a demo of the game that they need to develop before starting their implementation. Furthermore, students are encouraged to look at this demo to understand possible edge cases and behaviors they need to imitate.

With this in mind, I sought to create a study similar to Danas et al. (2017) but with a modification inspired by Wrenn and Krishnamurthi's work and the concept of a "demo" from CSCI 0150. Therefore, I changed Danas et al. (2017)'s study to include a satisfying instance of the model that students needed to develop at the beginning of the laboratory. Students in the experimental group would be exposed to the satisfying instance along with its local necessity. This satisfying instance was added with the hope of answering the question of how is looking at a given instance's local necessity before starting an assignment helpful to increase students' understanding of the model that they need to develop. Furthermore, as a secondary question, how is looking at the students' own satisfying instances' local necessity throughout the assignment, and comparing it to the initially shown instance's local necessity, helpful to identify over-constraints in their code.

In addition, because the users of Amalgam are not only supposed to be students, I was interested in running studies in the crowdsourcing platform Mechanical Turk. I wanted these studies to focus on Amalgam's ability to help users understand the constraints of a given model to gain a greater insight of the usability of the tool.

The main idea that influenced the design of my Mechanical Turk studies was a statement made in Nelson et al. (2017) on the benefits of Amalgam. Nelson et al. (2017) mentioned that when users observe a model's instances, it is really hard for them to retain and synthesize information from them to determine if the model has an over-constraint; however, they mention that Amalgam is able to point out under-constraints or over-constraints in the first instance seen [6]. With this in mind, I decided to design my Mechanical Turk studies in a way that would compare participant's ability to classify a model's constraints by seeing either multiple instances, or a smaller subset of instances with their local necessity.

3. Pedagogic Context

The course CSCI 1710: Logic for Systems at Brown University is an upper-level computer science course with a student enrollment of 66. The course focuses on proving properties about systems and programs through the use of formal methods tools. In order to register for this course, undergraduate students must have completed one of the three computer science introductory courses track, which includes a course on Algorithms and Data Structures. Graduate students can register under different conditions. With this in mind, most students that pursue CSCI 1710 have prior computing experience. Throughout the course of a semester, students need to complete eleven coding assignments, three coding projects, and eight labs (out of which one is optional). All of the enrolled students are grouped into weekly laboratory groups where they are then asked to complete the course's laboratories within a two-hour period. If students do not finish the laboratory in time, they have a week to complete it. During the laboratories, students can ask for help with the two teaching assistants present in the section. The course utilizes multiple formal method tools for the various assignments, some of these tools include Forge, Electrum[12], and Dafny [13].

Mechanical Turk is a crowdsourcing site where participants perform a given task for pay; given that the platform is virtual, it has the benefit of reaching people from around the world, including many technically savvy survey workers [14].

4. Implementing Amalgam in Forge

As mentioned previously, the course CSCI 1710 teaches and utilizes many programming languages, with the main one being Forge, a programming language developed by Prof. Nelson. Considering that Amalgam was originally implemented in Java as an extension of Alloy, I needed to re-implement Amalgam in Forge so I could recreate Danas et al. (2017)'s study in the course.

In order to re-implement Amalgam, I utilized Nelson et al. (2017) as a reference. This paper outlined the way that the tool worked along with its algorithm. In addition, Prof. Nelson gave me access to the Github repository containing the original implementation of Amalgam in Java.

The process of re-implementing Amalgam was divided into two main areas: (1) developing the tool to list out a tuple's provenance and (2) developing the tool to list the local necessity of a given instance. The former was implemented August-December 2020 and the latter was implemented January-February 2020 with the guidance of Prof. Nelson and the support of undergraduate student Abigail Siegel.

4.1 The Original Amalgam

The original implementation of Amalgam in Java is outlined in Nelson et al. (2017). For a detailed description of the algorithm and definitions please see [6]. In general, if a given literal `L` is locally necessary for a theory `T` in an instance[2] `M`, a set of provenances for `L` in `M` can be obtained by recursively evaluating the formulas in `T` in `M` and M^{L} (an alternate version of `M` with `¬L`) and recording the subformulas that led to `T`'s failure in M^{L}. A provenance for a literal `L` in `M` with respect to `T` is a set of sentences $\alpha_1, \ldots, \alpha_n$ where each alpha is true both in `M` and M^{L}, such that $T \cup \alpha_1, \ldots, \alpha_n \models_U L$ [6]. Note that entailment is subscripted; that is because Amalgam works with bounded satisfiability and so it is always with respect to a finite universe that all instances being considered must be bounded by.

In order to recursively evaluate the formulas in `T` in `M` and M^{L}, it is useful to have a desugaring function that instantiates and flattens formulas [6]. This is necessary because programming languages, like Alloy, have lots of syntactic sugar that can be ultimately reduced to `AND, OR,` and `NOT` operators. For example, the formula `A implies B` gets "desugared" into `(not A) or B`. In addition to removing syntactic sugar, every element constituting a formula has bounds, which allow for even more simplification.

Nelson et al. (2017) approached the recursive descent for every formula in one recursive tree that would first instantiate and flatten a given formula, get the bounds for that

[2] The source work refers to `M` as a model. To avoid inconsistency, we adopt the language used widely by the Alloy Community and call `M` an instance.

formula, and substitute values if needed. After the processing was complete, the algorithm converted provenance trees to provenances.

4.2 Implementing Provenance Feature

When I started drafting my approach to re-implement Amalgam, I reconsidered the design choice made by the authors of Amalgam to intertwine the desugaring and Amalgam descent processes [6]. While I knew that there were some benefits to that approach, I felt that segmenting these processes in different recursive trees would allow me to test each section independently.

With this in mind, my re-implementation of Amalgam required three recursive functions that operated on Forge's abstract syntax tree: Desugaring, Substitutor, and Lift-Bounds. Each of these recursive functions will be described in the following sections.

Given that I did not know how to work with Forge (or other parenthetical programming languages) before starting my implementation of Amalgam, I first implemented a prefix calculator that would work recursively to get exposed to the idea of recursion in Forge.

Desugaring: Explanation and Example

In order to desugar formulas and expressions, I created a recursive descent over Forge's abstract syntax tree that would take a formula or expression and match it to the possible values that it could have. After matching the given formula or expression, I was able to desugar it as needed.

There are cases where there is a need to get the formula's bounds or substitute values for variables in addition to removing syntactic sugar. For once, consider the formula `some Node`, where `Node` is a relation containing an upper bound of `{Node0, Node1}`. In order to desugar this formula, we would want `some var_x: union(upper-bound(Node)) | var_x in Node` to turn into a Forge constraint, where `upper-bound` is a helper function that gets the bounds of the atom. This is equivalent to,

```
some var_x: {Node0, Node1} | var_x in Node.
```

But, how did this work?

> The main purpose is to re-write `some Node` as the quantified formula `some fresh_variable : union(upper-bound(Node)) | fresh_variable in Node`
>
> In order to do this, we can follow the next steps.
>
> 1. Get the arity of the expression `Node`. The `Node` relation is arity 1.
> 2. In order to create a quantified formula, the algorithm needs to get the declaration variables. In order to get these, utilizing the arity, the algorithm accesses the column of the expression `Node` that it wants to look at.
>   a. For the given column, the algorithm wants to get the column's upper bounds. Getting the first column expression would result in `Node` and getting the upper bound of the `Node` relation would return `{Node0, Node1}`.
>   b. The algorithm then gets the union of the upper bounds, resulting in `{Node0, Node1}`.
>   c. The algorithm then creates a pair containing a fresh variable (`var_x`) and the union of the upper bounds got in step b: `(var_x, {Node0, Node1})`.
> 3. Now, the algorithm can make the formula `var_x in Node`.
> 4. This results in the quantified formula: `some var_x: {Node0, Node1} | var_x in Node.`

In this previous example, the algorithm had to access the upper bounds of the `Node` relation to be able to iterate through its values properly. In the case of desugaring set comprehension, for example, there is a need to substitute specific values into variables.

Lifting Bounds and Substituting

In order to access the bounds of a given expression, like we did in the previous example, I had to write another recursive descent that would take in a given expression and match it to its type (relation name, Int constant, quantified variable, etc.). After matching the expression to its type, the algorithm was able to get the bounds of the expression as required. For expressions such as set union, set minus, set intersection, within others, the algorithm could get their bounds by getting the bounds of the expressions' arguments.

Similarly, for the substitutor, I wrote a recursive descent for both formulas and expressions. This recursive descent would take either a formula or an expression and match on its type to substitute the given target variable with the desired value. Both `lift-bounds.rkt` and `substitutor.rkt` were tested thoroughly; having all of the recursive descents in their individual files was very useful to segment different functionalities and test them in detail. Additionally, each of these recursive descents had a respective helpers file which was also tested thoroughly.

The Amalgam Descent

While desugaring worked by calling the lift-bounds and substitutor functions, the code in charge of creating the actual provenances and doing the Amalgam descent lived in `amalgam.rkt`. The function in charge of building the set of provenances, `build-provenances`, has three main objectives. Taking in a tuple and a run statement as arguments, the function's first purpose is to create a formula `F` from the conjunction of predicates from the run statement. Afterwards, the second objective is to create an alternate instance with the same properties as the original run statement, except with the target tuple being flipped (i.e. making it False if it was originally True, and vice-versa). Finally, the last objective is to call the `amalgam-descent` function on `F` passing as arguments the original run statement, the alternate run statement, the original tuple, and a boolean representing the current sign of the formula.

Original Implementation of Amalgam Descent

The implementation for the Amalgam descent (**Y**) for the original Amalgam [6] can be seen in figure 4.

Figure 4. Algorithm for Amalgam Descent for Original Amalgam Paper

$$\mathcal{Y}(\bar{t} \in R) \equiv \begin{cases} \{\emptyset\} & \text{if } L = \bar{t} \in R \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{Y}(\alpha_1 \wedge \ldots \wedge \alpha_n) \equiv \bigcup\{\mathcal{Y}(\alpha_i) | \mathbb{M} \models \alpha_i, \mathbb{M}^{\mathbb{L}} \not\models \alpha_i\}$$

$$\mathcal{Y}(\rho_1 \vee \ldots \vee \rho_n \vee \gamma_1 \vee \ldots \vee \gamma_m) \equiv$$
$$\{\{\neg\rho_1, \ldots, \neg\rho_n\} \cup p | p \in (\mathcal{Y}(\gamma_1) \overset{\cup}{\times} \ldots \overset{\cup}{\times} \mathcal{Y}(\gamma_m))\}$$
$$\text{(where } \mathbb{M} \not\models \text{each } \rho_i \text{ and } \mathbb{M} \models \text{each } \gamma_j)$$

$$\mathcal{Y}(\neg\alpha) \equiv \mathcal{Y}_{\neg}(\alpha) \qquad \mathcal{Y}(\alpha) \equiv \mathcal{Y}(desugar(\alpha)) \text{ in all other cases.}$$

(Nelson et al. (2017))

This algorithm works as follows, where `L` is the target tuple:

- If the input formula is a literal and equal to the target tuple, then the algorithm returns a provenance set containing the empty provenance.
- If the input formula is a negation, `amalgam-descent` is called with the sign of the formula flipped.
- If the formula is a conjunction, then "the failure of any conjunct causes the overall formula to fail in $\mathtt{M}^{\mathtt{L}}$. The resulting provenance-set is, therefore, the union of all explanations for each conjunct's failure" [6].
- If the formula is a disjunction, then every disjunct must evaluate to false in $\mathtt{M}^{\mathtt{L}}$ and at least one disjunct should hold in `M`. This could also be represented as an implication where the left-hand side is made up of the negated disjuncts that are false in `M` and the right-hand side contains the formulas that were true in `M` and false

in M^{L} (labeled γ_i). The algorithm then calls the `amalgam-descent` on each of these γ_i to get the provenance for each one. These provenances are then combined with the cross-product and unioned with the conjunction of the negated ρ_i. A picture describing this step can be found on figure 5.

- If none of the previous steps hold, then there is a desugaring step and the algorithm calls the `amalgam-descent` on the desugared formula.

Figure 5. Explanation of 'OR' Step in Amalgam Descent

$$F = \underbrace{x_1 \vee x_2 \vee x_3}_{false\ in\ \mathbb{M}} \vee \underbrace{x_4 \vee x_5 \vee x_6}_{true\ in\ \mathbb{M}}$$

$$In\ \mathbb{M}^{\mathbb{L}}, x_1 \dots x_6\ are\ false$$

$$because\ x_1, x_2, x_3\ fail\ in\ \mathbb{M}, that\ implies\ that\ one\ of\ x_4, x_5, x_6\ holds\ in\ \mathbb{M}$$

$$\therefore F = \underbrace{\neg x_1 \wedge \neg x_2 \wedge \neg x_3}_{\alpha} \Rightarrow \underbrace{x_4 \vee x_5 \vee x_6}_{recur\ \mathcal{Y}\ on\ each\ of\ these}$$

Suppose that these are the provenances obtained:

$$\mathcal{Y}(x_4) = \{P_4^1, P_4^2\}$$
$$\mathcal{Y}(x_5) = \{P_5^1, P_5^2, P_5^3\}$$
$$\mathcal{Y}(x_6) = \{P_6^1\}$$

$$\therefore Provenances\ for\ \mathcal{L}\ in\ F = Cross\ Product\ of\ \mathcal{Y}(x_4), \mathcal{Y}(x_5), \mathcal{Y}(x_6)\ union\ \alpha$$
$$= \{\alpha \cup P_4^1 \cup P_5^1 \cup P_6^1,\ \alpha \cup P_4^2 \cup P_5^1 \cup P_6^1, \dots\}$$

*Note that the paper refers to the false x_i as ρ_i and the true x_i as γ_i

My Implementation of the Amalgam Descent

My implementation of the Amalgam descent was similar to that of the original Amalgam paper. However, the biggest difference between both algorithms is that my implementation utilizes a lazy stream to enumerate provenances. Given that the number of provenances for a literal `L` in a formula `F` can escalate exponentially through the cross product of the provenances of each γ_i, my algorithm provides users with one provenance at a time instead of computing all of the provenances at once, as the original Amalgam does.

For the example seen in figure 5, Forge's Amalgam would get the provenance stream for x_4, x_5, and x_6, take one provenance from each x_i, and create a provenance set by doing the union with alpha. The original implementation of Amalgam would get the provenance stream for x_4, x_5, and x_6, and store all of the possible provenances sets generated from the cross product. While the time complexity of my algorithm is the same as the original, my algorithm has an improved space complexity. When users ask for the "next" provenance, the

previous provenance is garbage collected. This is not the case for the original Amalgam, which requires space for all of the provenances computed.

For my Amalgam descent, I wanted to create a provenance tree made up of provenance nodes that I could access whenever needed. In order to create a tree of provenance nodes, I defined a `provenanceNode` struct in Forge and then created children of this struct called `contraLeaf`, `alphaLeaf`, `desugarStep`, `ANDProof`, and `ORProof`. Pseudocode for the amalgam descent function can be seen in figure 6.

Figure 6. Pseudocode for Amalgam Descent Function in Forge's Amalgam

```
amalgam-descent(fmla, orig-run, alt-run, L, currSign) {
      if fmla is (x_1 AND x_2 AND ... AND x_n):
            if currSign is positive: (handleAND x_1,x_2,...,x_n)
            else: (handleOR x_1,x_2,...,x_n)
      if fmla is (x_1 OR x_2 OR ... OR x_n):
            if currSign is positive: (handleOR x_1,x_2,...,x_n)
            else: (handleAND x_1,x_2,...,x_n)
      if fmla is a literal IN:
            if currSign is positive: contraLeaf(fmla)
            else: contraLeaf(not(fmla))
      if fmla is not (subf):
            (amalgam-descent subf orig-run alt-run L (not currSign))
      else:
            d_fmla = (desugar_formula fmla)
            prov_node = (amalgam_descent d_fmla orig-run alt-run L currSign)
            desugarStep(prov_node)
}
```

The `handleAND` and `handleOR` helpers work as follows:

1. `handleAND`: In this helper, every argument of the conjunction must have satisfied the original instance but not every argument must fail the alternate instance. For the arguments that do fail, the algorithm calls `amalgam-descent` and builds a provenance tree containing the nodes returned from each call. This helper returns the `provenanceNode ANDProof` with the provenance tree generated.
2. `handleOR`: In this helper, at least one argument of the disjunction must satisfy the original instance but all of the arguments must have failed the alternate instance. For the arguments that are true in the original instance, the algorithm builds a provenance tree from calling `amalgam-descent` on each argument. For the arguments that do not hold in the original instance (alphas), the algorithm creates an `alphaLeaf provenanceNode`. Afterwards, the helper returns an `ORProof provenanceNode` with the corresponding alphas and provenance tree.

In order to work with the provenance trees that I got from the `amalgam-descent`, I created a function `build-alphaset-stream` that takes in a `provenanceNode`. Pseudocode for this function can be seen in figure 7.

Figure 7. Pseudocode for Lazy Stream Provenance Function in Forge's Amalgam

```
(define (build-alpha-stream provNode)
      (if (provNode is contraLeaf(formula))
            // we found a contradiction but we have no alpha formulas yet;
            therefore, return stream containing the empty provenance
            (stream (list->set '())))
      (if (provNode is desugarStep(subtree))
            (build-alphaset-stream subtree))
      (if (provNode is ANDProof(subtrees))
            //each argument of the ANDProof is its own source of alpha-sets
            (apply stream-append (map build-alphaset-stream trees)))
      (if (provNode is ORProof(alphas, subtrees))
            // build lazy streams for each subtree
            substreams = (map build-alpha-stream subtrees)
            // build lazy stream of lists of provenances, where each list
            contains one provenance for each subtree
            stream = (stream-cross-product substreams)
            // Get the next n-element list of provenances and union with
            alphas
            (stream-map (λ (lst) (apply set-union (append alphas lst))),
            stream))
)
```

4.3 Implementing Local Necessity Feature

Before implementing Forge's Amalgam local necessity, I made an informal study with the CSCI 1710 Teaching Assistant staff utilizing the original Amalgam's local necessity feature to find areas of possible improvement.

For this informal study, the teaching assistants were given a simple colored undirected tree formal model in Alloy extracted from Nelson et al. (2017). The complete model can be seen in figure 8; note that the anti-reflexivity property is commented-out. Having this missing property meant the model had the under-constraint of allowing self-loops, but also, through the presence of the `treeAcyclic` predicate, had the over-constraint of prohibiting self-loops on models larger than one `Node`. Without being told about this bug, the teaching assistants were asked to take note of their thoughts about the tool, along with things that they found confusing or helpful.

Figure 8. Model for a Colored Undirected Tree with Missing Anti-Reflexivity Constraint

```
abstract sig Color {}
one sig Red extends Color {}
one sig Blue extends Color {}

sig Node {
  neighbors: set Node,
  color: one Color
}

fact undirected {
  neighbors = ~neighbors   -- symmetric
  -- INJECTED BUG
  --no iden & neighbors      -- anti reflexive
}

fact graphIsConnected {
  all n1: Node | all n2: Node-n1 |
    n1 in n2.^neighbors }

fact treeAcyclic {
  all n1, n2: Node | n1 in n2.neighbors implies
    n1 not in n2.^(neighbors-(n2->n1)) }

run {} for 3 Node
```

(Nelson et al. (2017))

My informal study demonstrated three major findings. These can be seen below:

- Two out of three participants reported that they found local necessity useful for identifying common types of bugs in Forge models.
- In Forge, you can choose a bound for the objects that your code models. For example, if you create a model for directed acyclic graphs, you can give a bound for the number of `Nodes` that you want in your instance. This bound is included along with the run statement for the model. If a model of directed acyclic graphs runs for a bound of `3 Nodes`, but the instance only displays `2 Nodes (Node0 and Node1)`, `Node2` can be called "unused". The original Amalgam would display local necessities for these "unused" `Nodes` as `unused<Node2>`. However, because the instance itself did not include `Node2`, two out of three participants reported that they found this "unused' display to be confusing. These participants thought that displaying this information distracted them from focusing on what was actually in the instance.
- One participant reported that they found the result of local necessity hard to read given that there was no visual connection between claims and the visualized instance, and the result is not sorted or grouped by signatures or relations.

With this information in mind, I moved into implementing the improved local necessity feature in Forge. Pseudocode for the algorithm can be seen in figure 9.

Figure 9. Pseudocode for Local Necessity in Forge's Amalgam

```
(define (get-locally-necessary-list orig-run instance-index)
      used_universe = all atoms used in instance
      relations = all relations from instance
      // get un-partitioned list of locally necessary tuples
      (define un-partitioned-list
            (apply append (for/list ([r relations])
                  (define name (Relation-name r))
                  valid_pairs = []
                  if r is not a sig or ints:
                        upper_bounds = (upper_bounds r)
                        for tuple in upper_bounds:
                              if (LN tuple) and (tuple in used_universe):
                                    append pair (tuple r) to valid_pairs)
            valid_pairs)
      //partition list into LN+ and LN- by evaluating in current instance
      (define-values (yes no) (partition (λ (pair)
               (define formula (tup in r))
               (evaluate orig-run 'unused formula)) valid_pairs))
      (cons yes no)
)
```

This feature returns two lists: `LN+` and `LN-`. `LN+` contains the tuples that need to be true in the original instance for the instance to be valid, and `LN-` contains the tuples that need to be false in the original instance for the instance to be valid. Per the feedback collected from the informal study, this implementation of local necessity is different from the original in that it does not display local necessity for "unused" nodes; in addition, Tristan Dyer helped me by integrating local necessity into the table and graph view of *Sterling*, the visualizer used for Forge, which he also developed [15, 16]. By doing this, I was able to show local necessity by signatures or relations.

Local Necessity of Original Amalgam versus Forge's Amalgam

Please see figures 10-16 for a comparison between the visualization for local necessity of the original Amalgam and Forge's Amalgam for the same satisfying instance of the model seen in figure 8.

Figure 10. Satisfying Instance

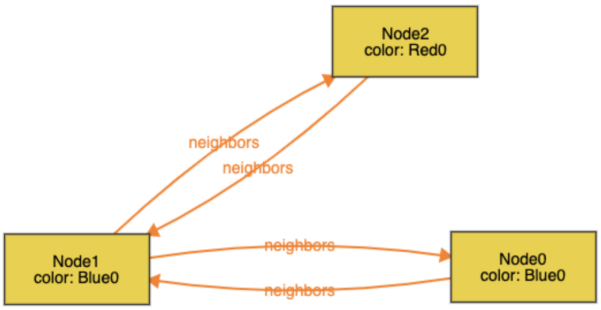
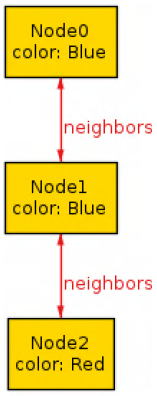
Forge's Visualizer	Alloy's Visualizer
	

Figure 11. Original Amalgam's Positive Local Necessity

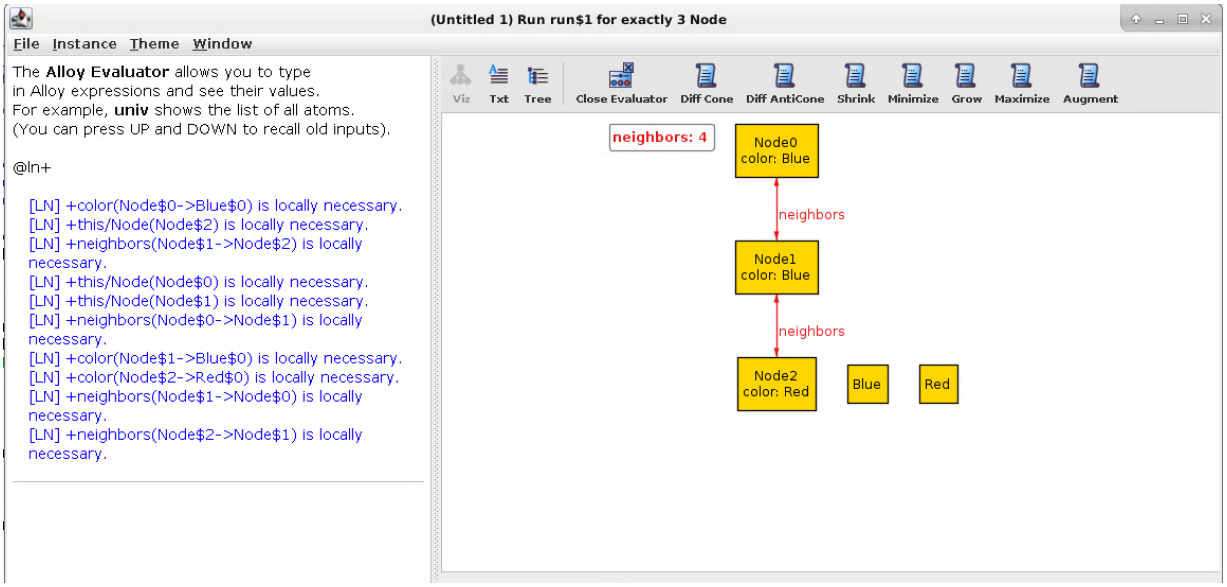


Figure 12. Original Amalgam's Negative Local Necessity

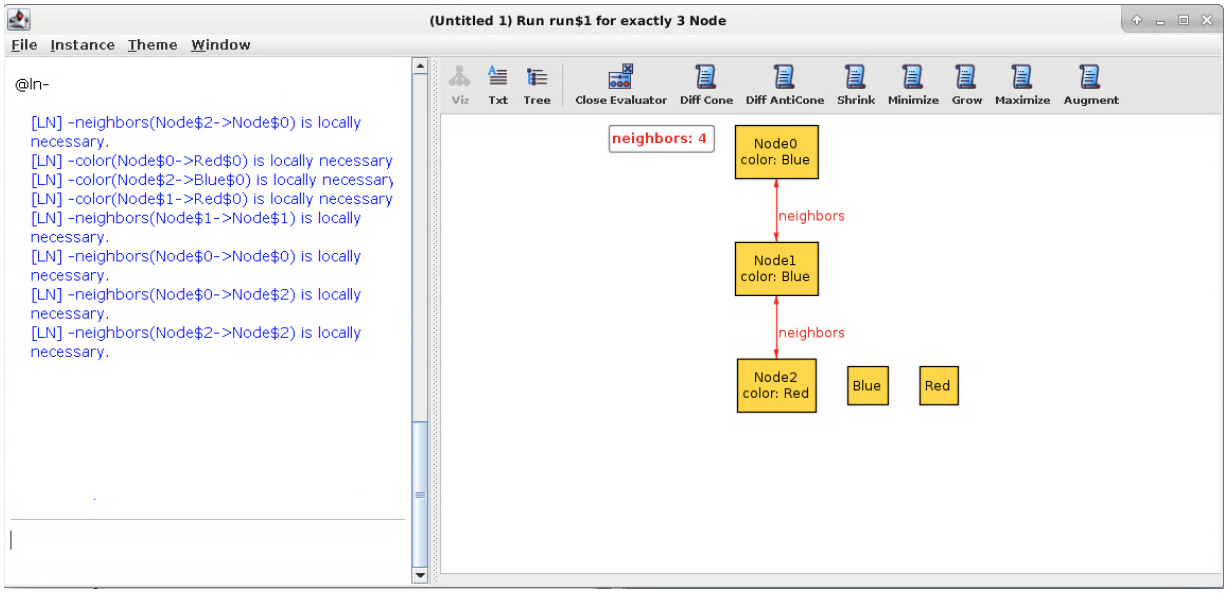


Figure 13. Forge's Amalgam Positive Local Necessity in Table View

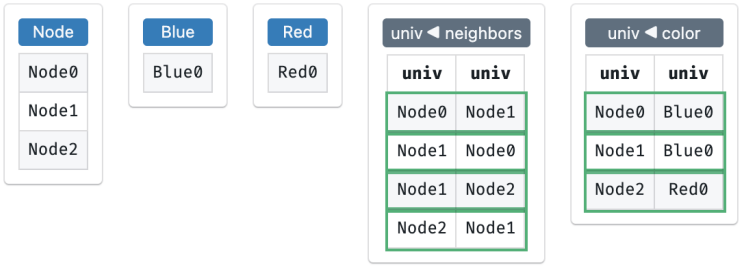


Figure 14. Forge's Amalgam Negative Local Necessity in Table View

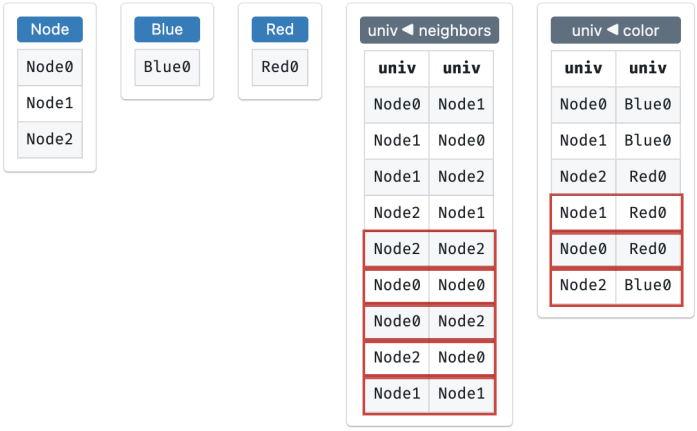


Figure 15. Forge's Amalgam Positive Local Necessity in Graph View

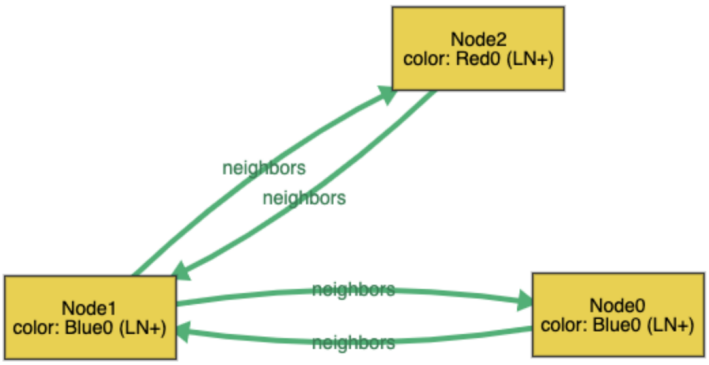
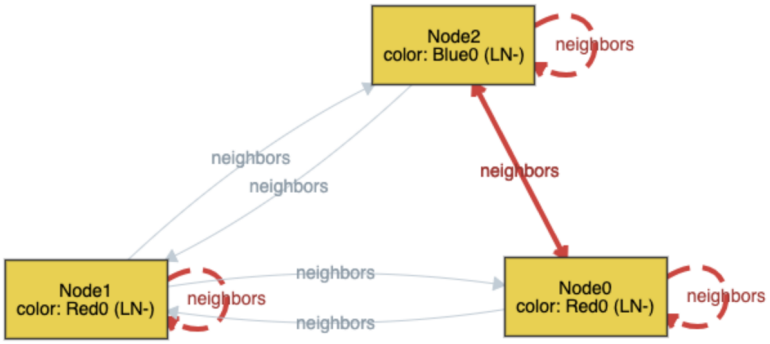


Figure 16. Forge's Amalgam Negative Local Necessity in Graph View



4.4 Benefits and Disadvantages of Implementing Amalgam in Forge

Over the course of my implementation of Amalgam, I found some benefits and disadvantages of developing such a tool in Forge. This section will go over these, as they are relevant for future developers that seek to incorporate and implement new tools into the language. It is important to note that throughout my implementation of Amalgam, I was working on the development branch of Forge which was not available to students and users of the language.

Benefits of Implementing Amalgam in Forge

As mentioned previously, the recursive descents needed to implement Amalgam were done over Forge's abstract syntax tree. Forge's abstract syntax tree is based on Ocelot's [17] abstract syntax tree which is made up of expressions that evaluate to relations and formulas that evaluate to booleans. In Forge, each formula or expression is represented with a `node` and followed by a specific hierarchy depending on it being a formula or an expression. Because Forge's abstract syntax tree allowed me to work with all formulas and

expressions freely, I was able to implement my recursive descents utilizing Racket's `match` function, which matches a given input formula or expression with its respective type. In addition, having the structure of Forge's abstract syntax tree allowed me to create new formulas and expressions very quickly and efficiently throughout the desugaring process.

Another big benefit of Forge is that it has the ability to define contracts for the inputs of a given function. This is extremely useful in debugging a program, especially a recursive descent where values can be altered erroneously in subsequent runs. Having contracts also facilitates the readability of the code for developers that might not be acquainted with the code for the tool.

Furthermore, because Forge is a language in the development phase, there is a lot of freedom to make changes to the current structure of the language to fit the functionality required for the tool. The implementation of Amalgam brought upon multiple changes in Forge, which will be described in detail in Section 4.5. Overall, having the ability to mold the language as a developer can be extremely helpful in implementing a new tool.

Disadvantages of Implementing Amalgam in Forge

While there is a lot of liberty and flexibility when implementing a tool in a developing language, there are also some limitations that come with it. Having the liberty as a developer to change the way that Forge does something in particular means that other developers have this same liberty, and some changes done by other developers can affect the work that someone else is doing. While implementing Amalgam, there were some cases where I had to change my approach because of a recent change in the code base, or where I would get errors in my code that were unrelated to my tool but were connected to a recent change done by another developer. Initially, differentiating between which errors were related to my tool or someone else's change was challenging, but with experience, this distinction became easier. In addition, being able to make changes to Forge or adding some type of functionality might also mean that other developers' changes need to be made before yours is possible. This can extend the timeline for the implementation of a given tool.

Through my implementation of Amalgam, I felt that one of Forge*'s* biggest weaknesses was the lack of documentation available for developers. Forge is a language focused mostly on students, and therefore, its documentation is mostly concentrated on the surface language which is exposed to the user[3]. For once, most of Forge's scriptable features were, and largely still are, undocumented and uncommented. As a consequence, there is a lot of confusion for developers about what is and isn't already available for use. Throughout my implementation, there were multiple times where I had to inquire about existing helpers or the way some helpers worked with someone that was more experienced with the language

[3] T. Nelson (personal communication, February 28, 2021) claimed that most of Forge's documentation is focused on the surface language which is exposed to the user.

than me. While there was always someone that I could contact about these questions, this generated a lot of dependency between the developer and the people who were more knowledgeable of the language. There also was a lack of documentation regarding testing in Forge. In order to write tests for Amalgam, I visited other files that had done testing and wrote my tests similarly.

Another limitation is that it is really hard to debug recursive descents if printlines are not enough for a specific program. While printlines can be helpful with small recursive descents, having a large recursive descent (which also goes into other recursive descents, as in my case) can make printlines confusing. There is a helper function for Racket called `debug-repl` that was extremely useful at the beginning of my implementation of Amalgam; however, an unknown change in Forge made `debug-repl` not work any longer, which made it hard to debug specific cases in the final testing process of Amalgam.

Another, smaller, complication with Forge is that there is some confusion regarding formulas and symbols. First, it is not the same thing in Forge to compare the formula `true` with `'true`. The former is a `node/formula/constant` value, while `'true` is just a symbol. While this might seem straightforward to follow, there were cases in Forge when things worked the other way around. For instance, in order to define a `node/expr/atom`, users need to define it utilizing `'` instead of just writing the name of the atom; therefore, users would do `(atom 'atomName)` instead of `(atom atomName)`. Similarly, there is some confusion regarding the functions that Forge allows and the functions that you need to import from Racket. For example, Forge does not allow the use of the greater-than and less-than operators; instead, you need to import these operators from Racket by adding the line `(require (prefix-in @ racket))` at the very top of the code file and utilize `@<` and `@>` whenever needed. This prefix requirement, or the functions that do not work well when developing in Forge, are not straightforward and can lead to unexpected problems.

Finally, the last limitation that I ran into with Forge was the ability to run alternate instances easily for building provenances and local necessities. Currently, the way that I am seeing if a given tuple is or isn't locally necessary is by building an alternate instance with the value of the tuple flipped. This alternate instance is then run by invoking Forge's solver directly, and seeing if the solver returns "satisfiable" or "unsatisfiable" (for more detail, refer to figure 9). However, invoking the solver every time that I want to run an alternate model makes the tool extremely slow (~30 seconds to a minute to run for a simple model running `5 Nodes`), and there is currently no other way to evaluate this alternate instance faster.

Suggestions to Improve Developer's Experience with Forge

Considering the previous sections, I came up with a list of suggestions to improve developers' experiences when implementing a new tool in Forge.

(1) Develop a point of reference to contact the Forge team for developer support. This point of reference can be a list-serv with the current members of the Forge developer team. This way, if a developer has a question regarding a specific change or helper function, anyone who knows the answer can reach out and answer questions. This would relieve the developer asking questions from reaching out to the same person every time and also might allow the entire team to be aware of the work that other developers are doing.
(2) Publish a weekly blog with the main changes that were done to Forge. This can be extremely helpful for developers that are implementing a tool in Forge and whose work might be affected by any recent changes. Publishing this blog entry with the changes, and even with some "expected" errors that might arise, can be very helpful.
(3) Develop a Forge debugger to allow developers to add breakpoints throughout their code.
(4) Develop detailed documentation for Forge's helpers, including a diagram explaining Forge's workflow for developers that are new to the language.
    (a) This documentation file should include guidance on available helpers and testing.
    (b) Along the same lines, this documentation can also include a list of common errors that developers might get in their code and the common causes for those errors.

4.5 Changes in Forge inspired by Amalgam

As mentioned previously, Amalgam inspired multiple changes in Forge. These changes were either new features needed to run the tool properly or bugs from Forge found along the way of my implementation.

Utilizing the `equals` Function for Testing Amalgam

In order to write tests for my desugaring function, I had to check for equivalence of formulas. In order to do so, I defined an input formula (`node/formula`) or expression (`node/expr`) and an expected output desugared formula (`node/formula`) or expression (`node/expr`). The output desugared formula/expression was compared to the output of calling my desugaring function with my input formula/expression. In order to check for equality between the expected and actual output, I used Racket's `equals` function. Initially, using the `equals` function in this way required first converting both elements to strings since the Forge language AST contained added information that interfered with direct comparison. With this in mind, as a temporary fix, I converted both formulas/expressions to

strings for comparison; however, Forge developers were able to modify the `equals` function to ignore the fields that interfered with direct comparison. Throughout this time, all of these changes to the `equals` function were made in the development branch of Forge; therefore, Forge users were not affected by them.

Forge Highlighting

The original Amalgam paper describes how the provenance feature highlights the top-level constraint that leads to the local necessity of a given tuple as well as subformulas that lead to the top-level constraints' failure [6]. Because highlighting was not available in Forge before implementing Amalgam, it became one of the biggest projects that developers worked on for Forge. Highlighting in Forge depended on the `node-info` field mentioned previously, and its addition to Forge was extremely helpful in finalizing Amalgam.

Introducing Local Necessity into Sterling

One of the biggest changes related to Forge and Amalgam was introducing information generated by Amalgam to Forge's visualizer, Sterling. By sharing the information that Amalgam generated with Sterling, students were able to see an instance's local necessity both on the graph and table view. This was a considerable improvement to the presentation of local necessity from the original Amalgam, as can be seen in figures 10-16.

When the first user study was run in CSCI 1710, local necessity was only shown as a part of the table view; however, for the round of studies in Mechanical Turk, I was able to test local necessity in the graph view of Sterling as well thanks to the support of Tristan Dyer.

Forge Logging

One of the biggest developments for Forge was the introduction of logging. Logging allows Forge developers to receive anonymous logs of student's runs, code snapshots, time-stamps, etc. Forge logging was extremely important to measure Amalgam's performance and usability during the user studies run in CSCI 1710. By having logging, I would be able to have access to both quantitative and qualitative metrics regarding the benefits of using local necessity versus not using it.

5. User Studies

5.1 Adaptation of Danas et al. (2017)

Structure of Study

The first laboratory for the course CSCI 1710 was adapted to run a study similar to the one in Danas et al. (2017). The course has four different sections for the laboratory, so two sections were the control group while the remaining two sections were the experimental group. The handouts for each of the sections can be found in the appendix. The model for the Connections of `KittyBacon` lab was slightly modified to the one presented in Danas et al. (2017). The complete model can be seen in figure 17.

Figure 17. Complete Model for `KittyBacon` Laboratory without `KittyBacon` in `FancyFelineFoundation`

```
#lang forge
option local_necessity on

sig Cat { friends : set Cat }
pred NoLonelyKittens { all c: Cat | some c.friends }
pred OutsideFriends { no iden & friends }
pred TwoWayStreet { friends = ~friends }
pred friendship {
    NoLonelyKittens
    OutsideFriends
    TwoWayStreet
}
one sig KittyBacon extends Cat { connectionsOf: set Cat }
pred connectionsOfKittyBacon {
    friendship
    KittyBacon.connectionsOf = KittyBacon.friends + KittyBacon.friends.friends
    + KittyBacon.friends.friends.friends - KittyBacon
}
pred ConnectedKittyBacon {
    connectionsOfKittyBacon
    Cat - KittyBacon = KittyBacon.connectionsOf
}
pred SuperConnected {
    connectionsOfKittyBacon
    Cat - KittyBacon in KittyBacon.^friends
}
pred ConnectedKittyBacon_equals_SuperConnected {
    ConnectedKittyBacon iff SuperConnected
}
check ConnectedKittyBacon_equals_SuperConnected for exactly 3 Cat
check ConnectedKittyBacon_equals_SuperConnected for exactly 4 Cat
check ConnectedKittyBacon_equals_SuperConnected for exactly 5 Cat

one sig FFF  { membersOfFoundation: set Cat }
pred FancyFelineFoundation {
    ConnectedKittyBacon
    FFF.membersOfFoundation = KittyBacon.connectionsOf
}
pred KittyBaconIsAFancyFeline {
    FancyFelineFoundation implies KittyBacon in FFF.membersOfFoundation
}
check KittyBaconIsAFancyFeline for exactly 3 Cat
pred seeAllCats {
    FancyFelineFoundation
    KittyBacon in FFF.membersOfFoundation
}
run seeAllCats for exactly 3 Cat
```

In this model, the `CoolCatClub` was renamed as the `FancyFelineFoundation,` and the predicate `KittyBaconIsCool` was renamed to `KittyBaconIsAFancyFeline.` In addition,

this model has the predicate `seeAllCats`, which was meant to encourage students to see a finalized version of their model.

The main differences between Danas et al. (2017) and my study are that my study makes the `FancyFelineFoundation` section of the laboratory mandatory and exposes students to a satisfying instance of the model before starting the assignment. In order to understand the satisfying instance at the beginning, both groups are given an overview of the laboratory and the relevant signatures and relations for context. The control group is shown the satisfying instance in the graph and table views of Sterling, and the experimental group is shown the instance in the graph view and positive and negative local necessities in the table view of Sterling. In addition, the experimental group was given a brief description of positive and negative local necessity. Both groups had been exposed to the concept of local necessity before the laboratories given that Prof. Nelson briefly explained it to all students in the lecture held previous to these labs.

In order to record the fix that students did to add `KittyBacon` to the `FancyFelineFoundation`, students were also asked to write a comment outlining their change.

Results

For this study, I wanted to collect multiple qualitative and quantitative measurements through Forge logging. I sought to collect: the total time that students took to complete the lab assignment, the time taken between the first run of `KittyBaconIsAFancyFeline` and the last run of the `seeAllCats` predicates, the time taken in between the first and last run of the `seeAllCats` predicate, the number of times that students ran their code since they ran the `KittyBaconIsAFancyFeline` predicate, the number of times that students ran their code since their first run of the `seeAllCats` predicate, the number of times that students ran predicates in the `FancyFelineFoundation` section, the number of changes that students made to their code after they first ran the `KittyBaconIsAFancyFeline` predicate, the number of times that students changed their code after they first ran the predicate `seeAllCats`, and the final snapshot of students' code. For students working with local necessity, I also wanted to collect data regarding the runs in which they utilized local necessity and the corresponding snapshot of their code at that time.

For this study, I attended all lab sections to answer any questions regarding local necessity or the lab overall. I also personally checked-off most of the students that attended the labs.

In order to analyze the results, I created a Python script that would go through all of the Forge logs and for a given student's anonymous email, return a report of all of the metrics that I wanted to collect. While I was analyzing the results, however, I noticed that a lot of student's logs were missing. Given that I had personally attended the lab and checked-off

multiple students, I knew that only a couple of students had not completed the lab during their section times and were going to complete it afterwards. However, of the logs that I had access to from the lab's experiment sections, there were only 8 valid logs out of 28. The remaining 20 logs were invalid, showing that students did not finish the lab even if they did. During the actual sections, I personally checked off a total of 18 students in the experimental group who had successfully completed the laboratory. According to the check-off sheet for the course CSCI 1710, 28 students from the experimental group were successfully checked-off, meaning that they finished the lab completely. Something similar, but to a smaller extent, happened with the control group, where I was not able to access the complete logs of 2 students even though every student in the control group was successfully checked-off.

Unfortunately, this means that many of the results for the lab study were lost because Forge logging was not working properly. Given the failure of logging, I made the decision to not consider the results collected from the laboratory, as they would be unreliable. I do not believe that the quantitative information from these logs can be trusted to make claims of the usefulness of local necessity.

The qualitative results from the lab are also not reliable given that I was missing multiple students' code snapshots that would allow me to see if their fix was correct or not. However, out of the 62 students that completed the lab, I personally checked-off 43 and they had all done the correct fix of modifying the definition of the `membersOf` the `FancyFelineFoundation` to be defined as the union of `KittyBacon` and `KittyBacon`'s `connectionsOf`. From the logs that I collected, 29 out of 29 valid logs from the control group show the correct fix to add `KittyBacon` to the `membersOf` the `FancyFelineFoundation`; furthermore, out of the 8 valid logs from the experimental group, all students did the right fix.

5.2 First Version of Mechanical Turk Studies

Given that the user study meant to test the usefulness of Amalgam's local necessity did not generate the results needed to create an analysis, I dedicated myself to developing user studies in the crowdsourcing platform Mechanical Turk. For these studies, I was interested in how Nelson et al. (2017) discussed that one of the benefits of Amalgam comes from helping users identify problems in their code in the first instance shown rather than requiring them to analyze and remember the constraints from multiple instances. Having this information in mind, I developed a study that would compare participant's ability to correctly classify the rules governing a given model without them seeing the model's code. In this study, the control group was shown multiple output instances' graphs from the normal Forge visualizer, while the experimental group was shown half the number of graphs as the control group, but each one came along with its positive and negative local necessity versions.

Structure of Studies

As mentioned previously, this experiment was set out to determine whether Amalgam's local necessity was actually helpful in aiding users to understand their model and its constraints better than seeing multiple instances from the visualizer. For this study, I created a formal model in Forge modeling a social network called `weHang`. The full model can be seen in figure 18.

Figure 18. Complete Model for Mechanical Turk Studies Modeling WeHang Social Network

```
1  #lang forge
2
3  option local_necessity on
4  sig NetworkUser {
5      follows: set NetworkUser,
6      isVerified: one Bool,
7      hasProfilePic: one Bool
8  }
9
10 abstract sig Bool {}
11 one sig Yes extends Bool {}
12 one sig No extends Bool {}
13 one sig Owner extends NetworkUser {}
14
15 pred noSelfFollower { no iden & follows }
16 pred verifiedNetworkUser {
17     all w:NetworkUser - Owner | ((#(follows.w) > 2) and w.hasProfilePic = Yes) iff w.isVerified = Yes
18 }
19 pred NetworkOwnerRestriction {
20     no follows.Owner
21     Owner.isVerified = Yes
22     Owner.hasProfilePic = Yes
23 }
24 pred NetworkOwnerFollowsAll {
25     all w:NetworkUser - Owner | w in Owner.follows
26 }
27 pred weHangSimulation {
28     noSelfFollower
29     verifiedNetworkUser
30     NetworkOwnerFollowsAll
31     NetworkOwnerRestriction
32 }
33
34 run weHangSimulation for exactly 4 NetworkUser, 5 Int
```

Line 3 adds the option to utilize local necessity for this model, Lines 4-8 define the properties of `NetworkUser`s. Each `NetworkUser` has the `follows` set consisting of the `NetworkUsers` that the `NetworkUser` follows. In addition, each `NetworkUser` has a boolean `isVerified` property representing whether it is verified or not. Along the same lines, each `NetworkUser` has a boolean `hasProfilePic` property representing whether it has or doesn't have a profile picture. Lines 10-12 represent the different values for `Bool`: `Yes` and `No`. Line 13 defines the `Owner` signature, which extends the properties of a `NetworkUser`. Line 15 defines the predicate `noSelfFollower`, stating that `NetworkUsers` can not follow themselves. Lines 16-18 define the `verifiedNetworkUser` predicate which states that for

all `NetworkUsers` except for the `Owner`, if the `NetworkUser` has more than 2 followers and has a profile picture, then the `NetworkUser` is verified. Lines 19-23 define the properties followed by the `Owner`; these include having no followers, always being verified, and always having a profile picture. Lines 24-26 contain the predicate `NetworkOwnerFollowsAll`, which specifies that the `Owner` should follow all `NetworkUsers` except itself. Finally, Lines 27-32 contain the `weHangSimulation` predicate. This predicate ensures that all previous predicates hold. Line 34 runs this predicate.

As mentioned previously, in order to understand which format was better for users to understand the constraints of a given model, I decided that I did not want to show the code to users, but rather only show them a subset of the satisfying instances of the model. At the beginning of both surveys, there was an initial questionnaire where participants were asked simple questions to test whether they were answering the survey randomly or not. This initial survey was added so that I could filter-out unthoughtful responses for my analysis. After the initial survey, participants were prompted to see multiple satisfying instances and asked to write in their own words what they thought were the rules governing the social network. Afterwards, participants were asked to classify a list of rules governing the social network between the ones that were true and the ones that were false; these rules were a translation made from the constraints of the model in figure 18 into English.

Before running my study with a large set of people, I ran five iterations of it with smaller groups (2-10 people). These iterations allowed me to find common points of confusion or areas of improvement for the survey. Some of the changes that I made between iterations were:

(1) Initially, participants were shown a list of the governing rules of the social network and asked to select the rules that were true out of the list. I noticed that when participants were given access to the entire list, they would only select one or two rules, especially the rules in the middle of the screen. Therefore, I decided to make the change of separating each rule into an individual question.

(2) I noticed that if a page had multiple questions, participants would start answering really accurately and continue to answer the last questions almost randomly, sometimes even contradicting previous responses. Therefore, I made each question be in its own individual page so participants could focus more on the question at hand.

(3) In my survey, participants are shown multiple graphs and then asked to answer a set of questions. I noticed by running my survey with my housemates that remembering the contents of the graph for the final questions was really hard, and they started guessing instead of going back to see the graph in question. Therefore, I added the picture of the same graph to every question.

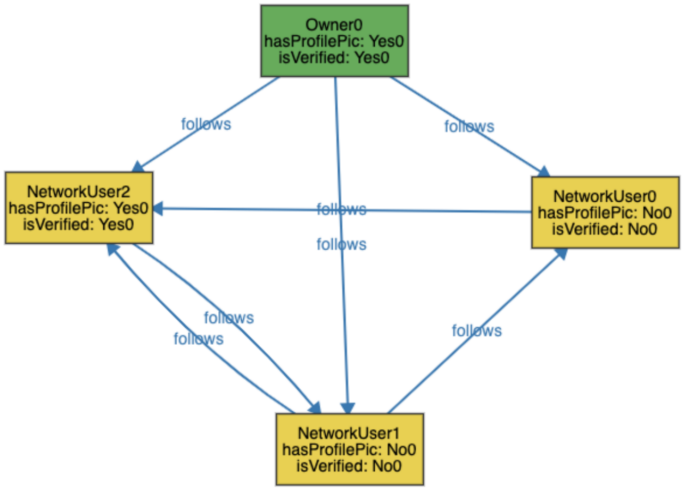
(4) In an open-ended question in the survey, some participants reported being confused about the graphs being shown and their components. Therefore, I added a description of the graphs and their components before starting the survey.
(5) The social network has an `Owner`. At the beginning, I had named the `Owner` `NetworkOwner`; however, I noticed that participants were getting confused between `NetworkOwner0` and `NetworkUser0`. Therefore, I changed the name of the `Owner` to just `Owner`.
(6) I decided to add a question at the end of the survey asking participants if they took any notes throughout the study, and in case they did, which notes they wrote down. I added this question because I was curious about comparing whether participants from the control group wrote down more notes than participants from the experimental group or vice-versa.

After all of these iterations and improvements, I ran the final version of my study with 25 participants for the control group and 27 participants for the experimental group. Initially, I had the purpose of having the same sample size for both the control and experimental group; however, for the experimental group, I ran into the issue of not getting responses from Mechanical Turkers even though the survey had been published for over a week. In order to get the results that I was missing to reach 25 responses, I had to publish the survey once again for only two respondents. At the time that I started my data analysis, however, I received the two missing responses from the original publication, and therefore worked with 27 responses instead of 25.

Both the experiment and control group were exposed to an initial instance of the social network model along with a description of the instance's components. The instance, along with the description included in both surveys, can be seen in figure 19.

Figure 19. Introduction to Social Network Graphs for Both Studies

In the following social network graph, `Yes0` stands for "Yes" and `No0` stands for "No". "`hasProfilePic`" is an attribute describing whether a `NetworkUser` or `Owner` has a profile picture, and "`isVerified`" is an attribute describing whether a `NetworkUser` or `Owner` is verified. The "`follows`" arrows represent following a given `NetworkUser` in the social network. The outgoing arrow from `NetworkUser0` to `NetworkUser2` means that `NetworkUser0` follows `NetworkUser2`. The followers of `NetworkUser2` are `NetworkUser0`, `NetworkUser1`, and `Owner0`.



Afterwards, participants moved on to the initial questionnaire and were asked six questions to test their understanding of the instance itself: (1) How many boxes are there for `NetworkUsers`?, (2) How many boxes are there for `Owner`?, (3) Who does `NetworkUser1` follow?, (4) Who does `Owner0` follow?, (5) Who has a profile picture (meaning that their "`hasProfilePic`" attribute is set to `Yes0`)?, (6) Who is verified (meaning that their "`isVerified`" attribute is set to `Yes0`)?. Each question was multiple-choice, and answers to these questions were used to filter out unthoughtful responses.

Additionally, the experimental group had a second questionnaire to test their understanding of positive and negative local necessities. Given the same graph as above, they were given its respective positive and negative local necessity graphs along with a description of the graphs' components. The graphs, along with their descriptions, can be seen in figures 20-22.

Figure 20. Introduction to Local Necessity Graphs for Experimental Group

Consider the graph seen in the previous section. Let's call it `graphA`.

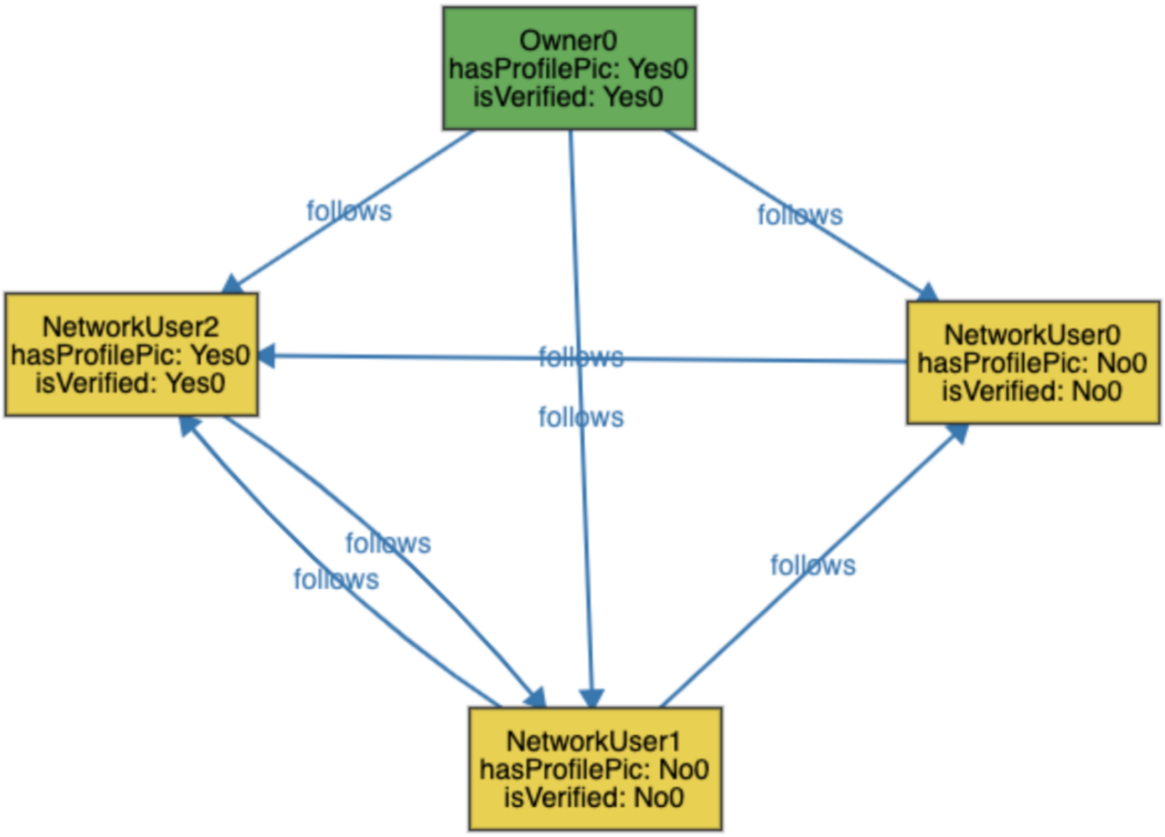


Figure 21. Introduction to Positive Local Necessity Graphs for Experimental Group

Here we see `graphA` with some green lines. The green lines mean that if you tried to remove that line, `graphA` would become invalid. As an example, if we were to remove the green "`follows`" line from `Owner0` to `NetworkUser2`, the graph would become invalid. Removing the green lines makes `graphA` invalid because there are other things in `graphA` that depend on that green line being there. For example, imagine that `Owner0` always needs to follow `NetworkUser2`; this would mean that removing the "`follows`" line would make the graph invalid). For the attributes of `NetworkUsers` and `Owners` (`isVerified` and `hasProfilePic`), there are no lines, but the annotation "`LN+`" represents the same idea as a green line. For example, `Owner0`'s "`hasProfilePic`" field is "`Yes0 (LN+)`". This means that you can not change the "`hasProfilePic`" field to "`No0`" without making `graphA` invalid because there are other things in `graphA` that depend on `Owner0`'s "`hasProfilePic`" field to be "`Yes0`". A gray line can be removed without making the graph invalid because there is nothing else in the graph that requires that line to be there. This means that we could remove the "`follows`" arrow from `NetworkUser1` to `NetworkUser0` without making the graph invalid.

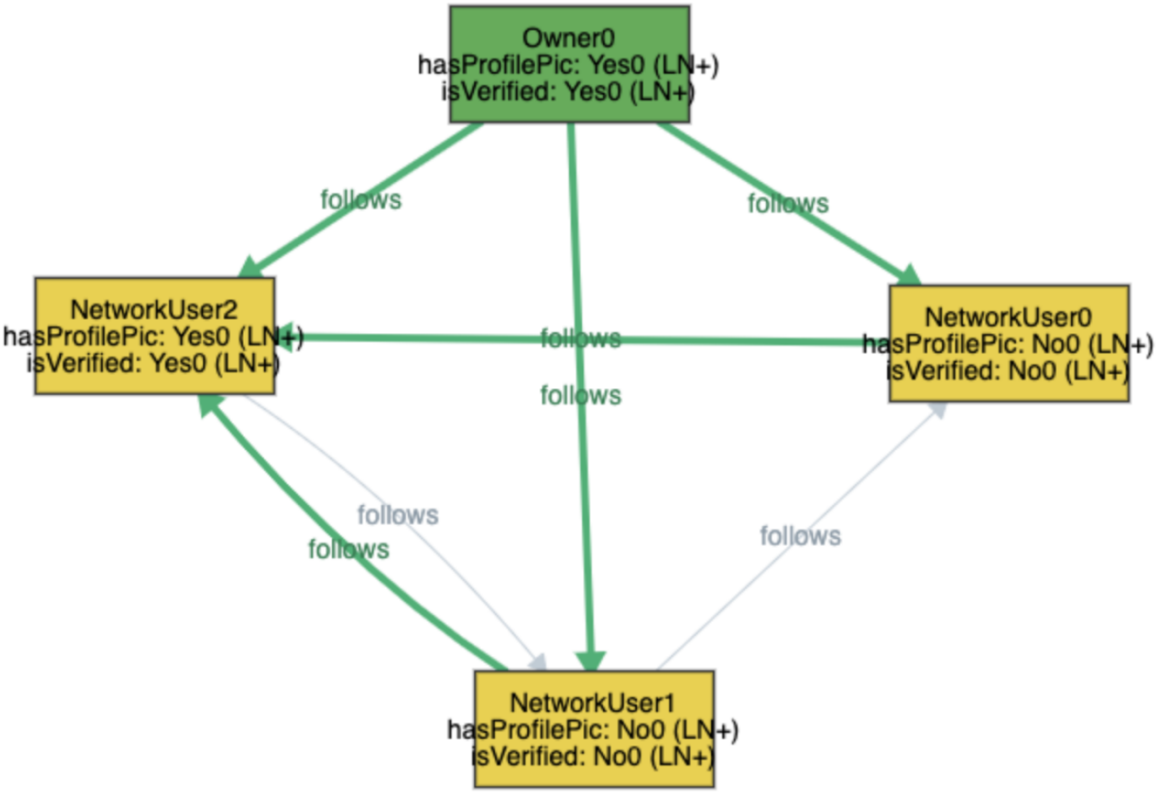
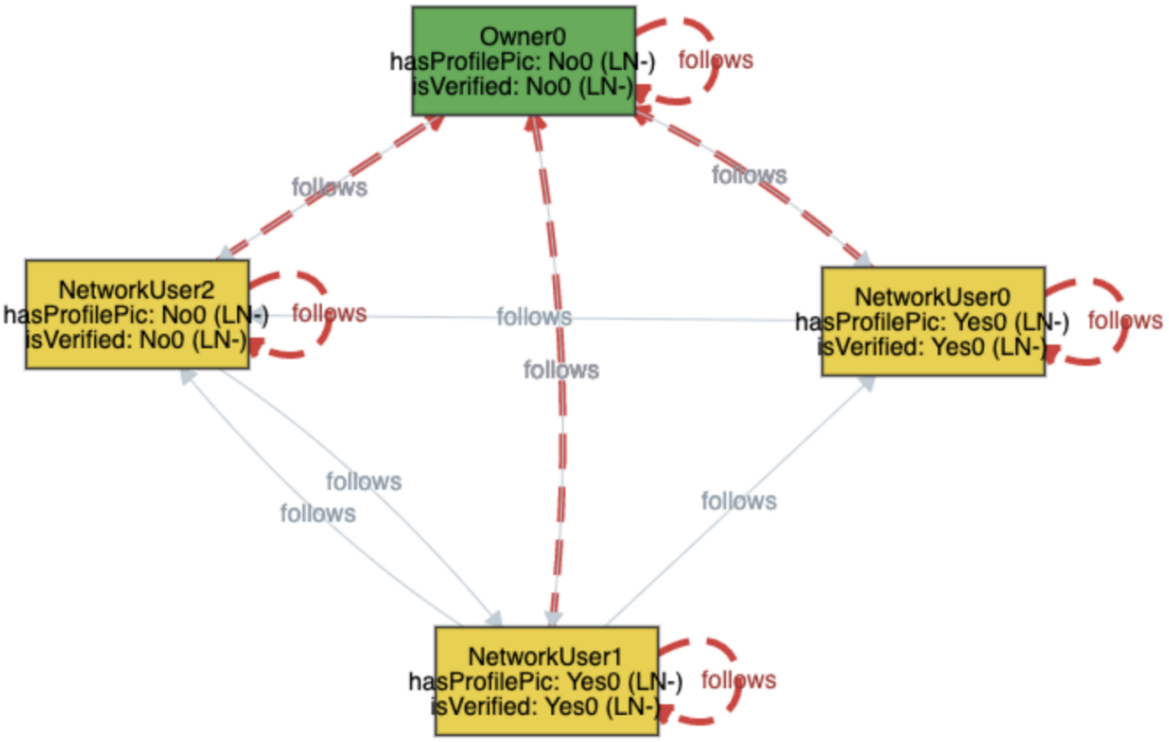


Figure 22. Introduction to Negative Local Necessity Graphs for Experimental Group

Here we see `graphA` with some red-dotted lines. The red-dotted lines mean that if you tried to add any red-dotted line, `graphA` would become invalid. As an example, if we were to add the red-dotted "`follows`" line from `NetworkUser1` to `Owner0`, `graphA` would become invalid. Adding the red-dotted lines makes `graphA` invalid because there are other things in `graphA` that depend on that red-dotted line not being there. For the attributes of `NetworkUsers` and `Owners` (`isVerified` and `hasProfilePic`), there are no lines, but the annotation "`LN-`" represents the same thing as the red-dotted line. For example, `Owner0`'s "`hasProfilePic`" field is "`No0 (LN-).`" This means that making `Owner0`'s "`hasProfilePic`" field "`No0`" makes the graph invalid because there are other things in `graphA` that depend on `Owner0`'s "`hasProfilePic`" field to not be "`No0`". A gray line represent existing facts in `graphA`.

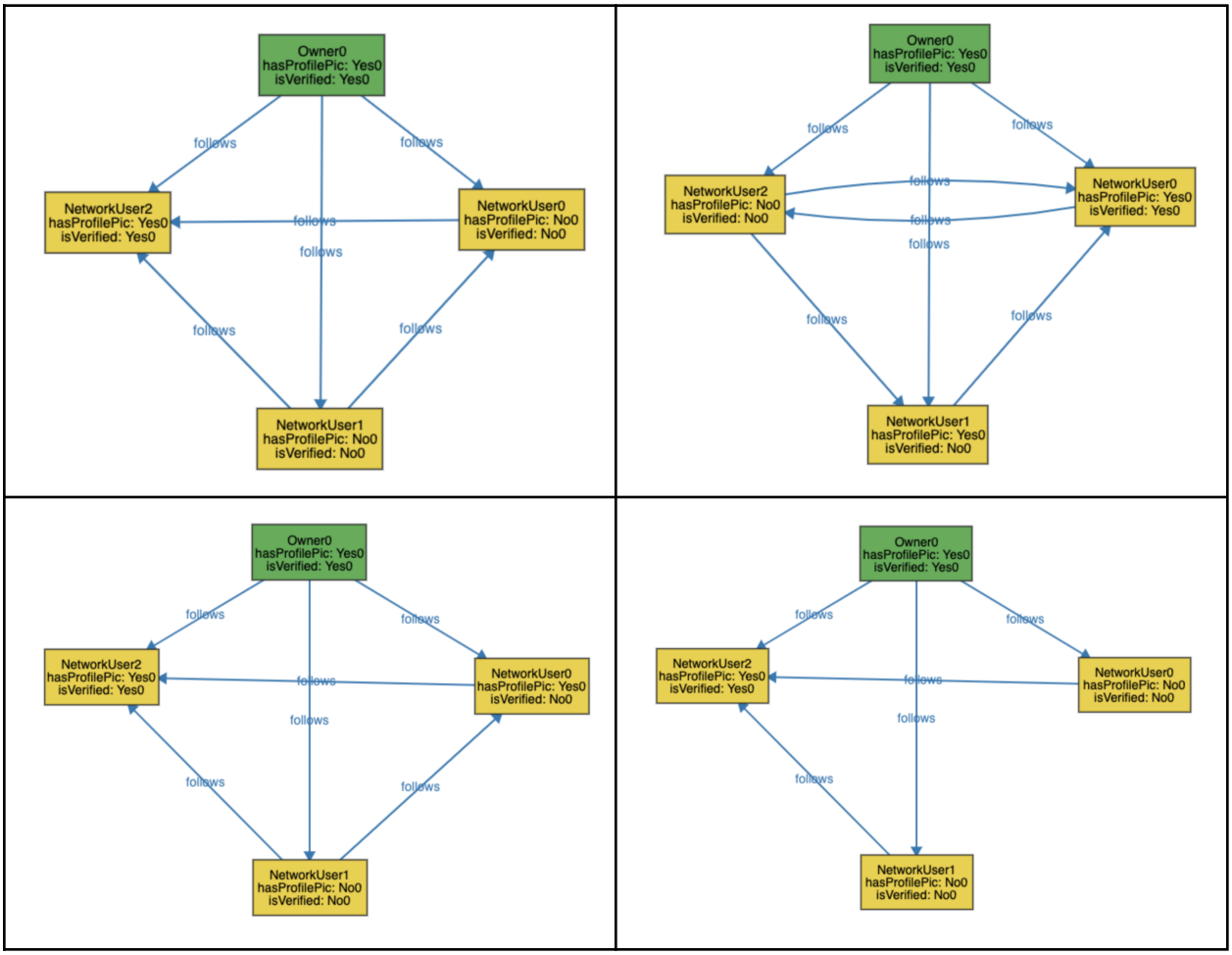


Afterwards, the experimental group was asked four questions to test participant's understanding of the local necessity graphs. These questions were (1) "What would happen if we added the red-dotted "follows" line from NetworkUser1 to NetworkUser1?", (2) "What would happen if we removed the green "follows" line from NetworkUser1 to NetworkUser2?", (3) "What would happen if we changed the "isVerified" property of NetworkUser2 from Yes0 to No0?", and (4) "What would happen if we removed the gray "follows" line from NetworkUser2 to NetworkUser1?". For each of these questions,

participants had to choose between two response options: “the graph would be invalid”, and the “graph would be valid”.

After going through these questions, both the control and experimental groups continued to see instances of the model. The control group saw eight graphs of the model, and the experimental group saw a subset of four graphs seen by the control group along with the graph’s positive and negative local necessity. The graphs seen by each group can be seen in figure 23 and 24.

Figure 23. Control Group Graphs



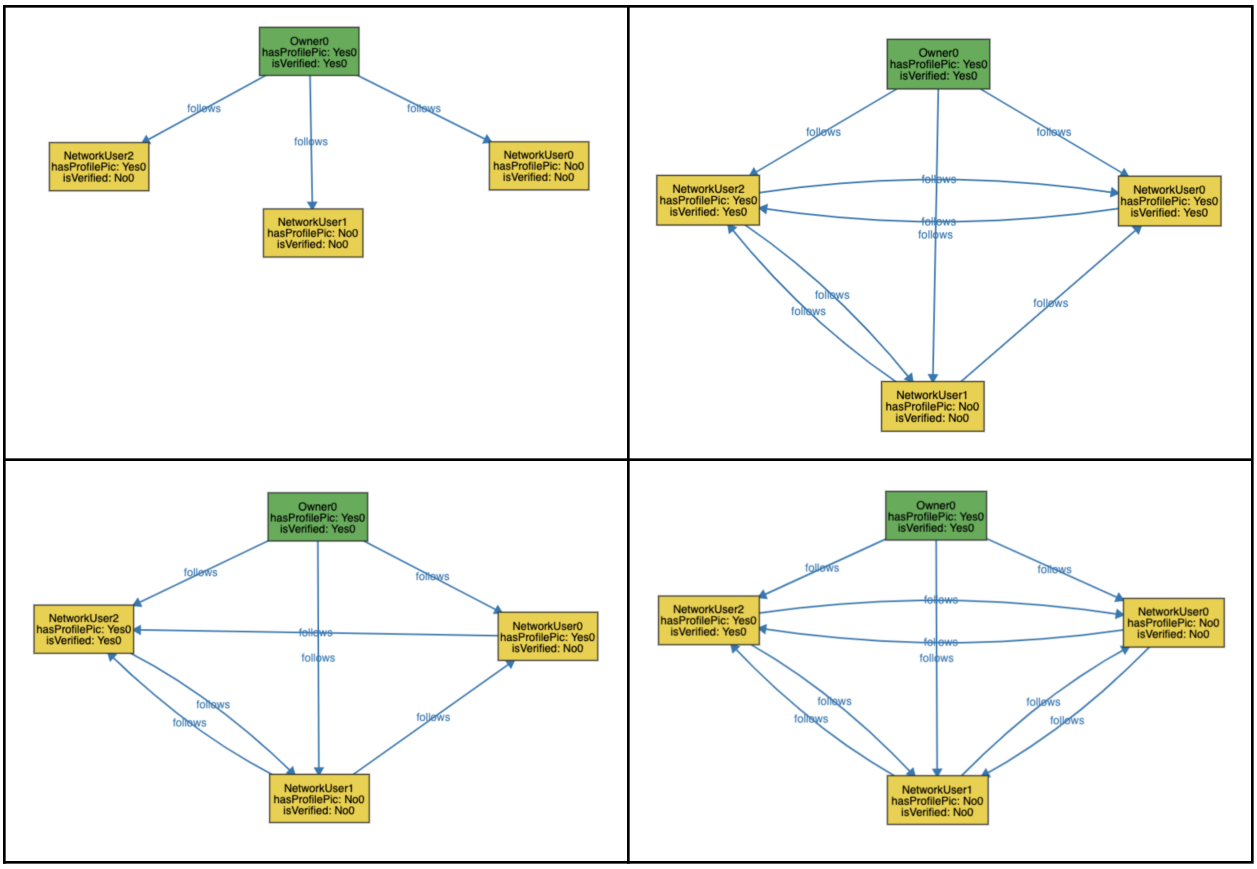
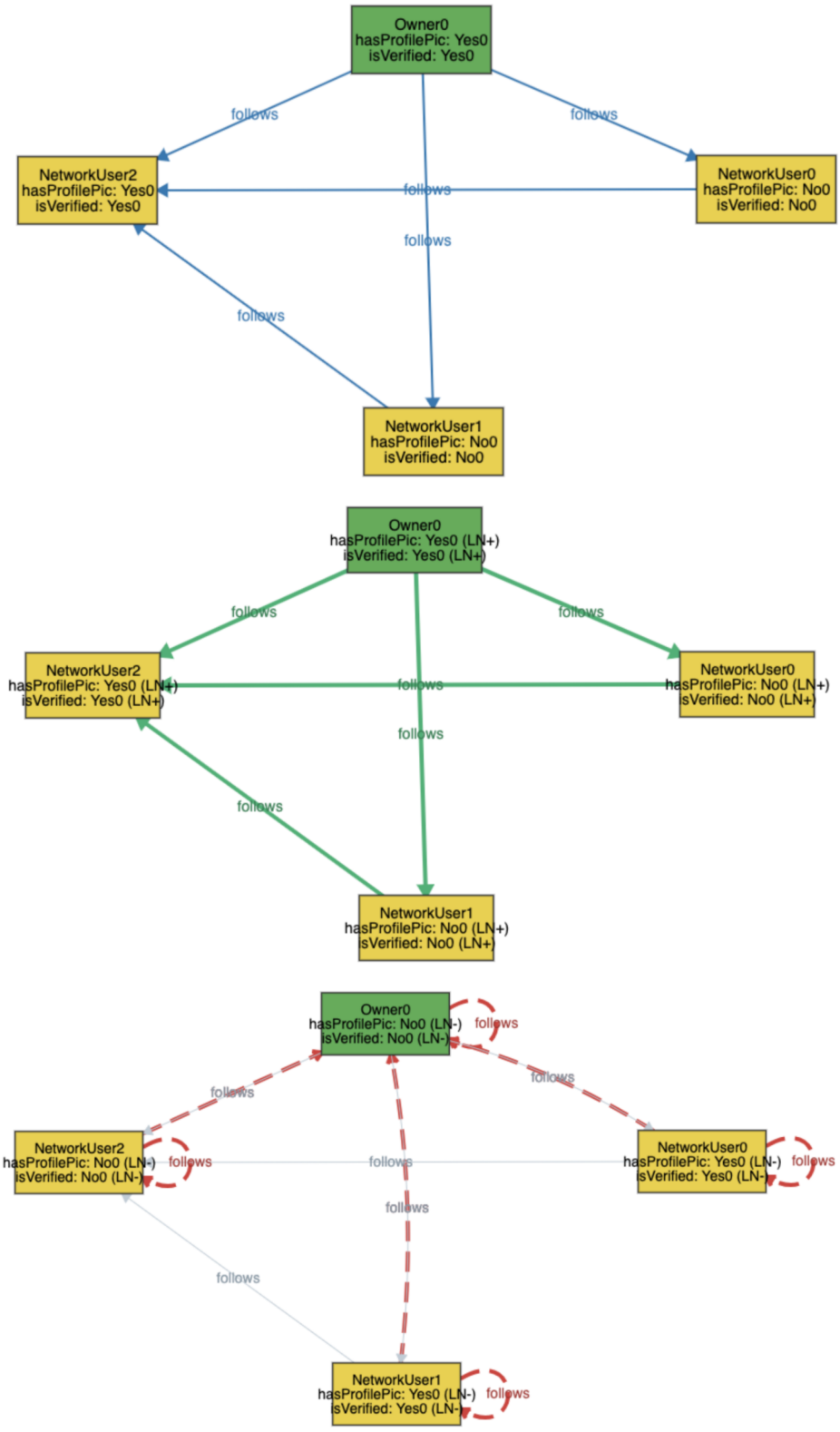

Owner0
hasProfilePic: Yes0
isVerified: Yes0
follows
NetworkUser2
hasProfilePic: Yes0
isVerified: No0
NetworkUser1
hasProfilePic: No0
isVerified: No0
NetworkUser0
hasProfilePic: No0
isVerified: No0
NetworkUser2
hasProfilePic: Yes0
isVerified: Yes0
NetworkUser0
hasProfilePic: Yes0
isVerified: Yes0

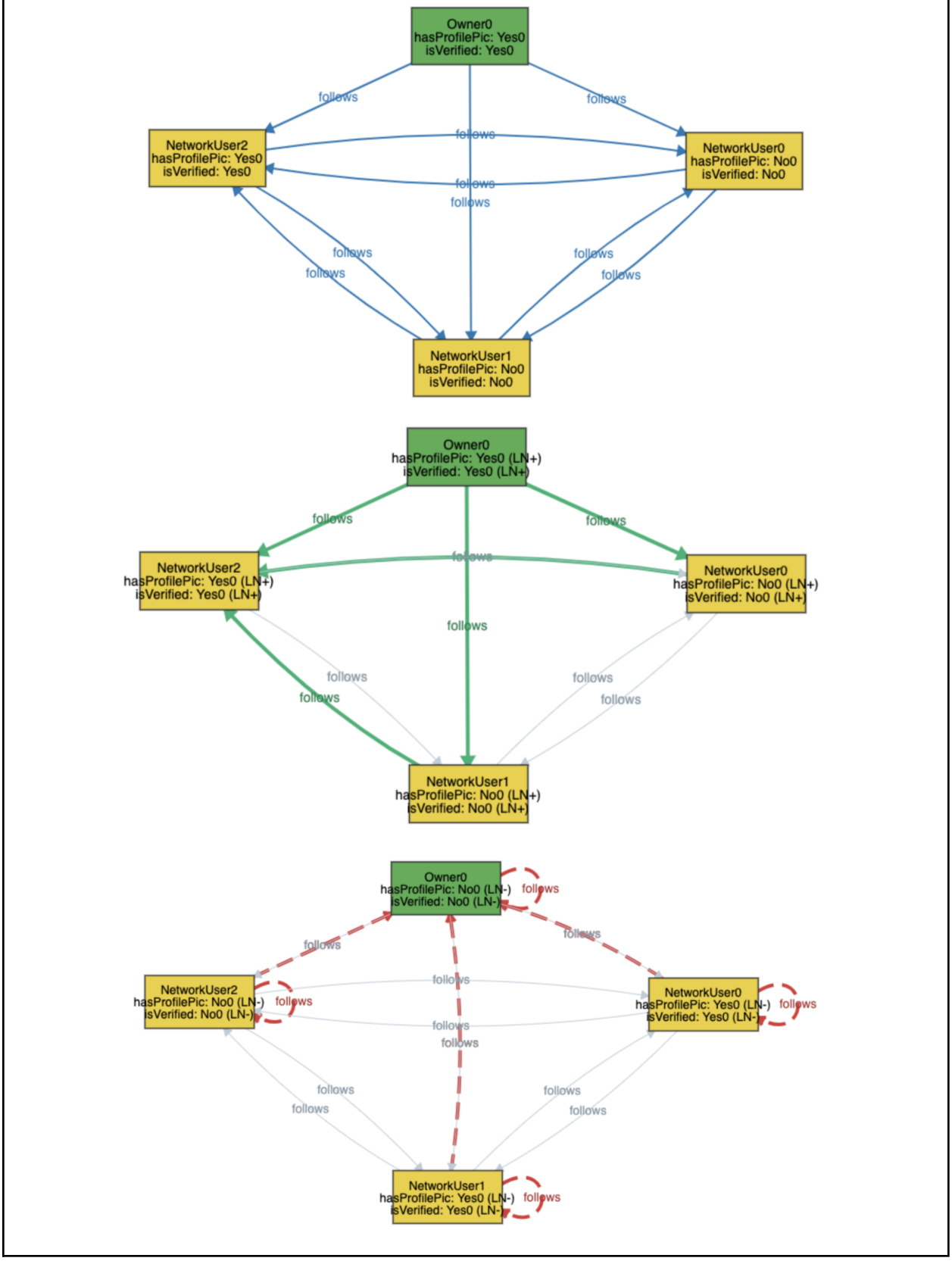
Figure 24. Experimental Group Graphs

Graph 1

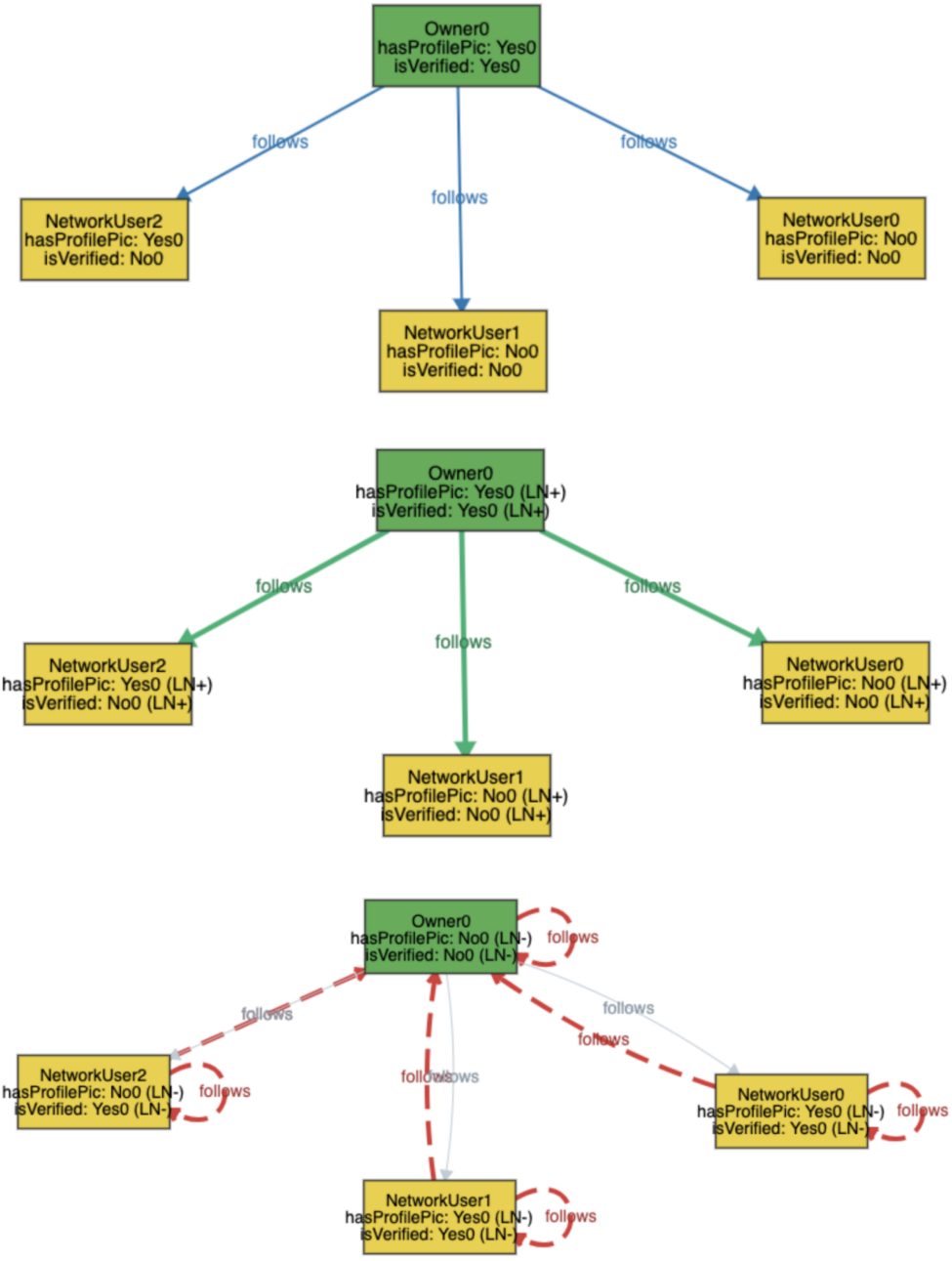
Graph 2


Owner0
hasProfilePic: Yes0
isVerified: Yes0
follows
follows
NetworkUser2
hasProfilePic: Yes0
isVerified: Yes0
follows
NetworkUser0
hasProfilePic: No0
isVerified: No0
follows
follows
NetworkUser1
hasProfilePic: No0
isVerified: No0
Owner0
hasProfilePic: Yes0 (LN+)
isVerified: Yes0 (LN+)
follows
follows
NetworkUser2
hasProfilePic: Yes0 (LN+)
isVerified: Yes0 (LN+)
follows
NetworkUser0
hasProfilePic: No0 (LN+)
isVerified: No0 (LN+)
follows
follows
NetworkUser1
hasProfilePic: No0 (LN+)
isVerified: No0 (LN+)
Owner0
hasProfilePic: No0 (LN-)
isVerified: No0 (LN-)
follows
follows
follows
NetworkUser2
hasProfilePic: No0 (LN-)
isVerified: No0 (LN-)
follows
follows
NetworkUser0
hasProfilePic: Yes0 (LN-)
isVerified: Yes0 (LN-)
follows
follows
follows
NetworkUser1
hasProfilePic: Yes0 (LN-)
isVerified: Yes0 (LN-)
follows

Graph 3


Owner0
hasProfilePic: Yes0
isVerified: Yes0
NetworkUser2
hasProfilePic: Yes0
isVerified: Yes0
NetworkUser0
hasProfilePic: No0
isVerified: No0
NetworkUser1
hasProfilePic: No0
isVerified: No0
Owner0
hasProfilePic: Yes0 (LN+)
isVerified: Yes0 (LN+)
NetworkUser2
hasProfilePic: Yes0 (LN+)
isVerified: Yes0 (LN+)
NetworkUser0
hasProfilePic: No0 (LN+)
isVerified: No0 (LN+)
NetworkUser1
hasProfilePic: No0 (LN+)
isVerified: No0 (LN+)
Owner0
hasProfilePic: No0 (LN-)
isVerified: No0 (LN-)
NetworkUser2
hasProfilePic: No0 (LN-)
isVerified: No0 (LN-)
NetworkUser0
hasProfilePic: Yes0 (LN-)
isVerified: Yes0 (LN-)
NetworkUser1
hasProfilePic: Yes0 (LN-)
isVerified: Yes0 (LN-)
follows

Graph 4


Owner0
hasProfilePic: Yes0
isVerified: Yes0
follows
follows
follows
NetworkUser2
hasProfilePic: Yes0
isVerified: No0
NetworkUser1
hasProfilePic: No0
isVerified: No0
NetworkUser0
hasProfilePic: No0
isVerified: No0
Owner0
hasProfilePic: Yes0 (LN+)
isVerified: Yes0 (LN+)
follows
follows
follows
NetworkUser2
hasProfilePic: Yes0 (LN+)
isVerified: No0 (LN+)
NetworkUser1
hasProfilePic: No0 (LN+)
isVerified: No0 (LN+)
NetworkUser0
hasProfilePic: No0 (LN+)
isVerified: No0 (LN+)
Owner0
hasProfilePic: No0 (LN-)
isVerified: No0 (LN-)
follows
follows
follows
follows
follows
NetworkUser2
hasProfilePic: No0 (LN-)
isVerified: Yes0 (LN-)
follows
NetworkUser0
hasProfilePic: Yes0 (LN-)
isVerified: Yes0 (LN-)
follows
NetworkUser1
hasProfilePic: Yes0 (LN-)
isVerified: Yes0 (LN-)
follows

After seeing these graphs, both groups were asked to write down, in their own words, the rules of the social network that generated them. After doing that, both groups were shown rules that might hold for the social network model, and asked to select "True" if they thought the rule held or "False" otherwise. The list of rules shown to participants can be seen below.

1. `NetworkUsers` do not need to follow other `NetworkUsers`
2. If `NetworkUserA` follows `NetworkUserB`, then `NetworkUserB` must follow `NetworkUserA`
3. The `Owner` needs to follow all `NetworkUsers`
4. `NetworkUsers` can't follow themselves
5. `NetworkUsers` users can have no followers
6. `NetworkUsers` need two things for the "`isVerified`" attribute to equal "`Yes0`". The first is to have a profile picture ("`hasProfilePic`" field set to "`Yes0`"). The second is to be followed by the `Owner` and 2 or more other `NetworkUsers`.
7. `NetworkUsers` need only one thing for the "`isVerified`" attribute to equal "`Yes0`". That is, they need to be followed by the `Owner` and 2 or more other `NetworkUsers`. The "`hasProfilePic`" field does NOT matter (it can be set to `Yes0` or `No0`).
8. The `Owner` always has a profile picture

Finally, participants from both groups were asked if they took any hand-written notes during the exercise and what sort of information they wrote down.

Results

In order to analyze the results of this study, I filtered out the responses that had more than one incorrect answer for the initial questionnaire of the control and experimental groups. This resulted in 19 valid responses for the control group and 25 valid responses for the experimental group. The distributions for correctly classifying each rule for both the control and experimental groups can be seen in figures 25-26.

Figure 25. Results for Control Group and Experimental Group for Rules # 1-4

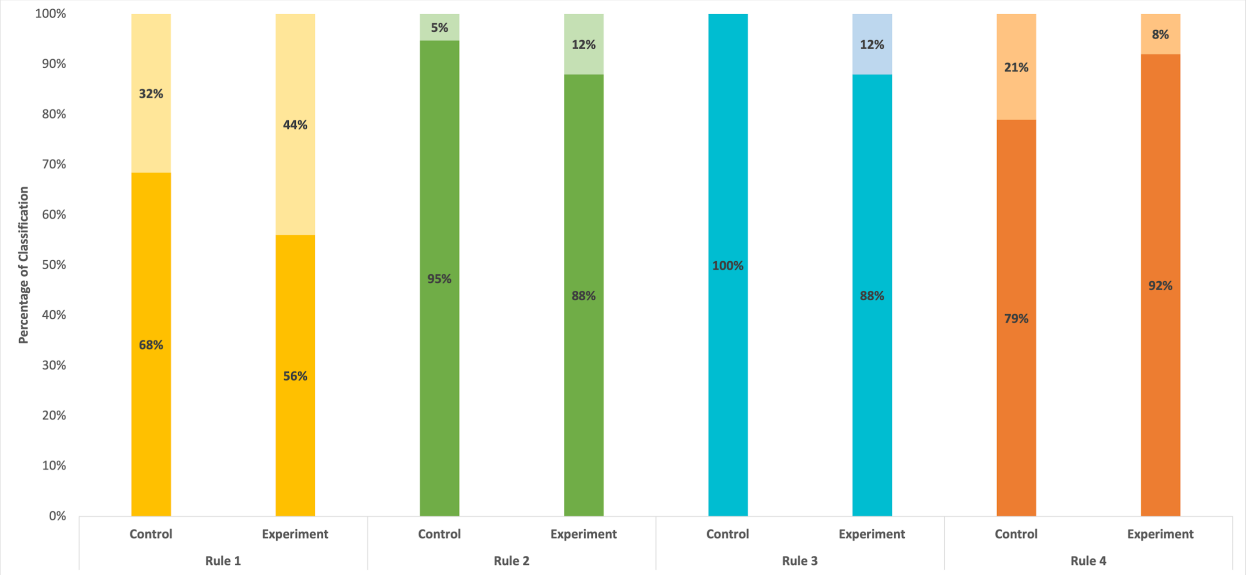
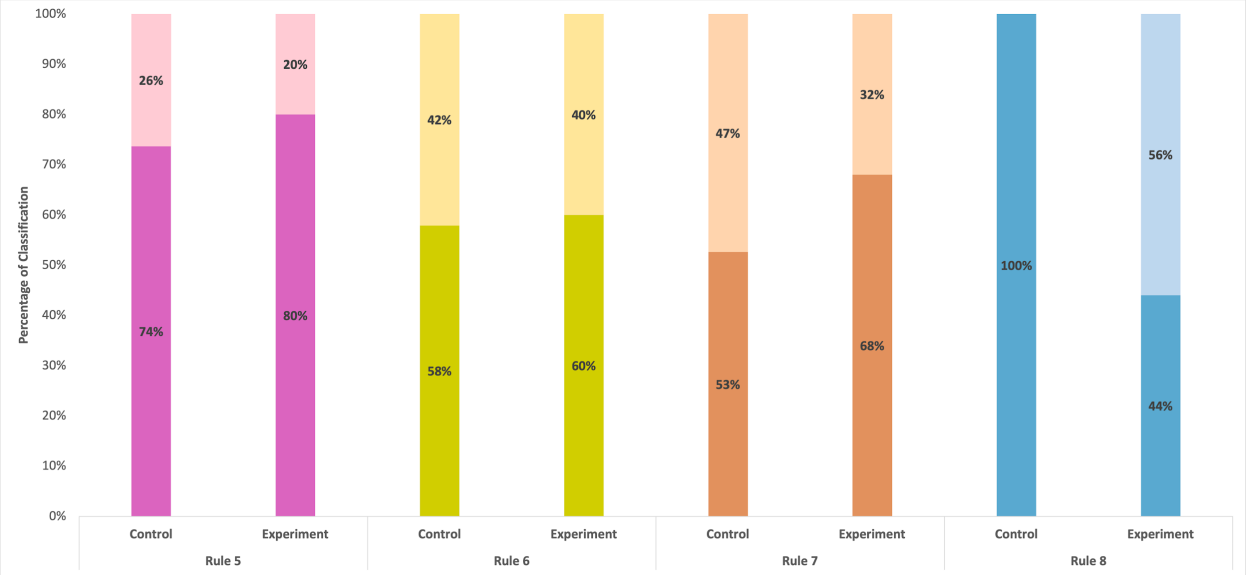


Figure 26. Results for Control Group and Experimental Group for Rules # 5-8



After seeing the previous table graphs, it is noticeable that the results favor the control group in 4 rules and the experimental group in 4 rules. In order to measure whether these results were statistically significant, I performed a Chi-Squared analysis for each question. Additionally, I calculated Hedge's G to get the effect size of the strength of the relationship between participants' performance and the type of graph participants observed.

The Chi-Squared analysis was done by comparing whether participants correctly classified a given rule and the type of graph seen. For example, for the first rule, I have the distribution in figure 27.

Figure 27. Example Chi-Squared Distribution for Rule #1

Correctly Classified the Rule?	**Type of Graph Seen**		
	Regular Graphs	**Local Necessity Graphs**	**Total**
Yes	13	14	27
No	6	11	17
Total	19	25	44

After calculating the Chi-Squared distribution for each rule with a significance level of 0.05, I got the following results:

Figure 28. Chi-Squared Analysis Results

Rule Number	**Chi-Squared Result**	**Calculated P-value**	**Conclusion**
1	0.702506593	0.401942722	Results are not statistically significant
2	0.592842105	0.441322194	Results are not statistically significant
3	2.446829268	0.117762531	Results are not statistically significant
4	1.561735919	0.211411229	Results are not statistically significant
5	0.245201238	0.620474488	Results are not statistically significant
6	0.019793072	0.88811654	Results are not statistically significant
7	1.07545465	0.299716612	Results are not statistically significant
8	**15.60533333**	**7.80342E-05**	**Results are statistically significant**

As seen in the previous table, the only result that was statistically significant was the one related to rule number 8. Rule number 8 states that "The `Owner` always has a profile picture". For this rule, all of the participants in the control group with valid responses

correctly classified this rule; however, of the valid responses of the experiment rule, only 11 out of 25 participants correctly classified it. This finding was very interesting given that all of the graphs of the experimental group stated that it was locally necessary for the `Owner`'s `hasProfilePic` to be `Yes0`. I questioned why the results were split almost 50-50 for this question if the local necessity for this field remained the same throughout the four graphs. I hypothesized that the culprit was the visualization of local necessity itself. There is a possibility that participants were confused by the use of `Yes0` and `No0` for the `Owner`'s `hasProfilePic` field in the positive and negative local necessity graphs, respectively. This would explain why the results for the experimental group are almost split 50-50; half of the group focused on the `Yes0` of the positive local necessity graph while the other half of the group focused on the `No0` of the negative local necessity graph. The use of `Yes0` and `No0` for `Owner`'s `hasProfilePic` field can be seen in figure 29.

Figure 29. Comparison of Negative and Positive Local Necessity's Owner Boxes

Positive Local Necessity Graph Owner Box	Negative Local Necessity Graph Owner Box
	

Keeping this in mind, I wondered whether there was another way of modifying the graphs with local necessity to be clearer to participants. After talking to a member of the Forge development team that works on Sterling, I was told that no other modification could be made at the time being without requiring significant effort. Therefore, I made my own version of positive and negative local necessity graphs utilizing Microsoft Powerpoint to represent the graphs that I would suggest for the representation of local necessity in Sterling. This led to a secondary Mechanical Turk study that I will go over in section 5.3.

Going back to the data analysis, for the effect size I originally considered using Cohen's D. However, because I ended up having samples of different sizes for my control and experimental groups, I decided that utilizing Hedges' G to measure effect size would be better since Hedges' G provides a measure of the effect size weighted to the relative size of the sample. Since Hedges' G requires the average and standard deviation for both the control and experimental group, I had to find a way to get the average of all of the questions without the average just being the percentage of correct classifications. Therefore, I calculated a score for each participant of my user studies, where each correct classification of the model's rules gave one point. Therefore, a score of 0-8 was assigned for each participant. Utilizing these scores, I was able to then get the average score for the group and the standard deviation of these scores.

Figure 30. Average Scores and Standard Deviations for Control and Experimental Groups

Control Group	Experimental Group
Average Score: 6.263157895 Standard Deviation: 1.240165995	Average Score: 5.76 Standard Deviation: 1.42243922
Hedges' G: 0.373443. Small Effect Size	

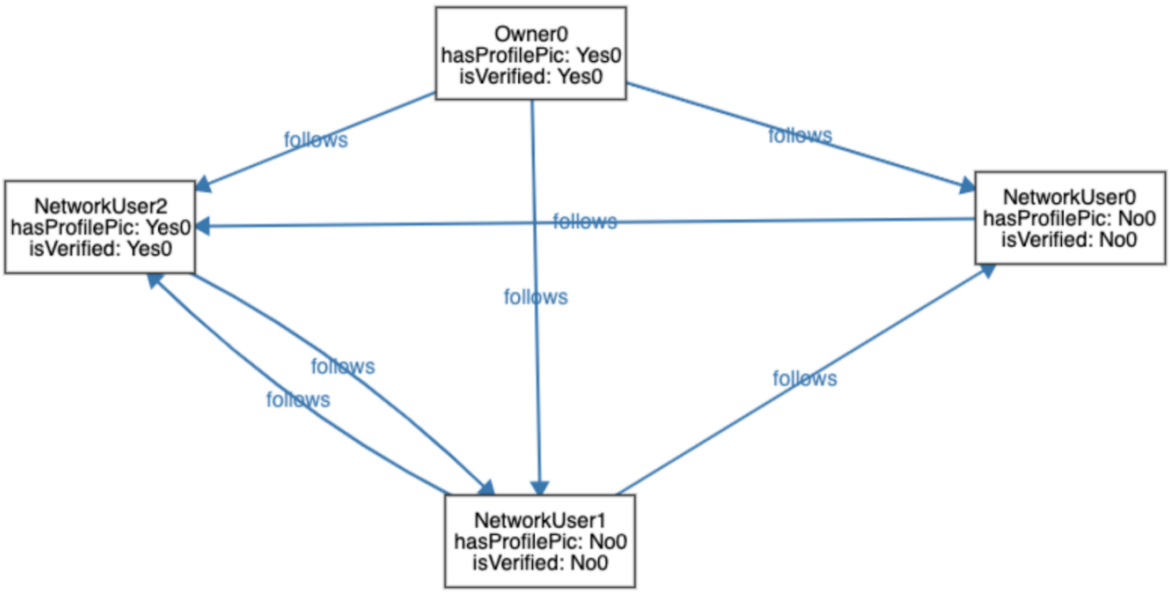
5.3 Second Version of Mechanical Turk Studies

For this study, I wanted to modify the positive and negative local necessity graphs to help participants understand local necessity better. In addition to thinking about what I would change, I also wanted to consider the opinion of people who were exposed to these graphs without having any previous context. With this in mind, I sent the original survey to some friends and asked them what confused them about the graphs. One of my friends reported that the green background of the `Owner`'s block made her feel as if the `Owner`'s fields were always valid, independent of the positive or negative local necessity graphs. The green background for the `Owner`'s block became even more confusing to her when positive local necessity was introduced with green arrows. Another friend reported that the annotations "`LN+`"and "`LN-`" in the attribute fields were very confusing, and their meaning wasn't clear enough. With these points in mind, I modified both the original version of the graphs and the positive and negative local necessity graphs.

Structure of Studies

This secondary study remained relatively the same as the previous version of the experimental group; the only thing that changed was the format of the graphs and the wording of the explanation of the positive and negative local necessity graphs. This study was run for 25 participants. The graph shown to users before the initial survey changed to the graph seen in figure 31.

Figure 31. Initial Graph Shown to Participants of Updated Survey



The new graphs for negative and positive local necessity, along with their descriptions, can be seen in figures 32-34.

Figure 32. Updated Local Necessity Description for Experimental Group

Consider the graph seen in the previous section. Let's call it `graphA`.

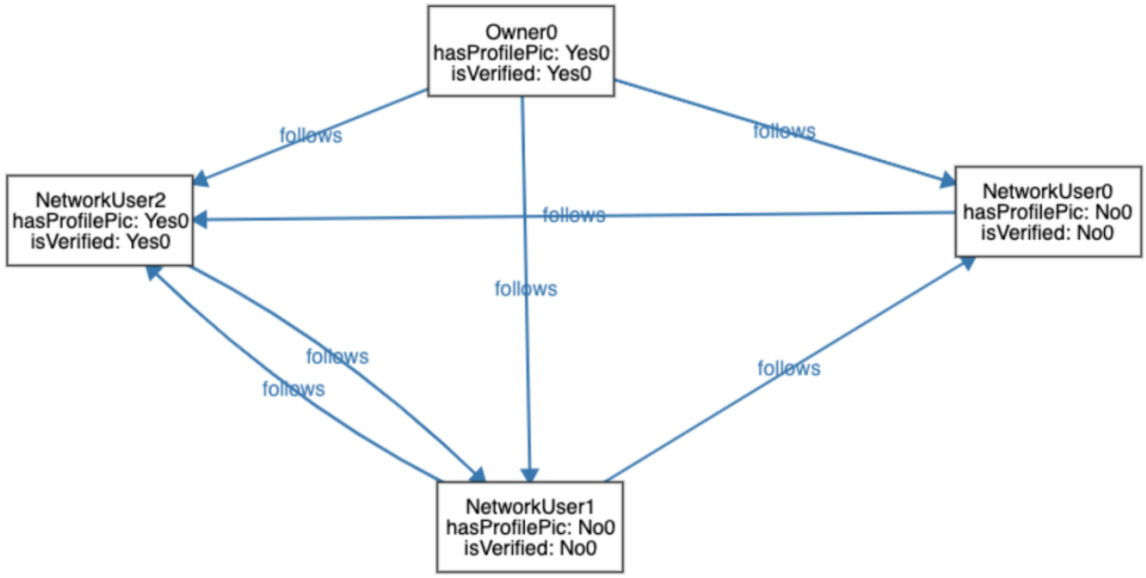


Figure 33. Updated Positive Local Necessity Description for Experimental Group

Here we see `graphA` with some green lines. The green lines mean that if you tried to remove that line, `graphA` would become invalid. As an example, if we were to remove the green "`follows`" line from `Owner0` to `NetworkUser2`, the graph would become invalid. Removing the green lines makes `graphA` invalid because there are other things in `graphA` that depend on that green line being there. For the attributes of `NetworkUsers` and `Owners` (`isVerified` and `hasProfilePic`), if the attribute is highlighted in green it means that changing that value will make the graph invalid. For example, `Owner0`'s "`hasProfilePic`" field is "`Yes0`" highlighted in green. This means that changing the "`hasProfilePic`" field to "`No0`" makes `graphA` invalid because there are other things in graphA that depend on `Owner0`'s "`hasProfilePic`" field to be "`Yes0`". A gray line can be removed without making the graph invalid because there is nothing else in the graph that requires that line to be there. This means that we could remove the "`follows`" arrow from `NetworkUser1` to `NetworkUser0` without making the graph invalid.

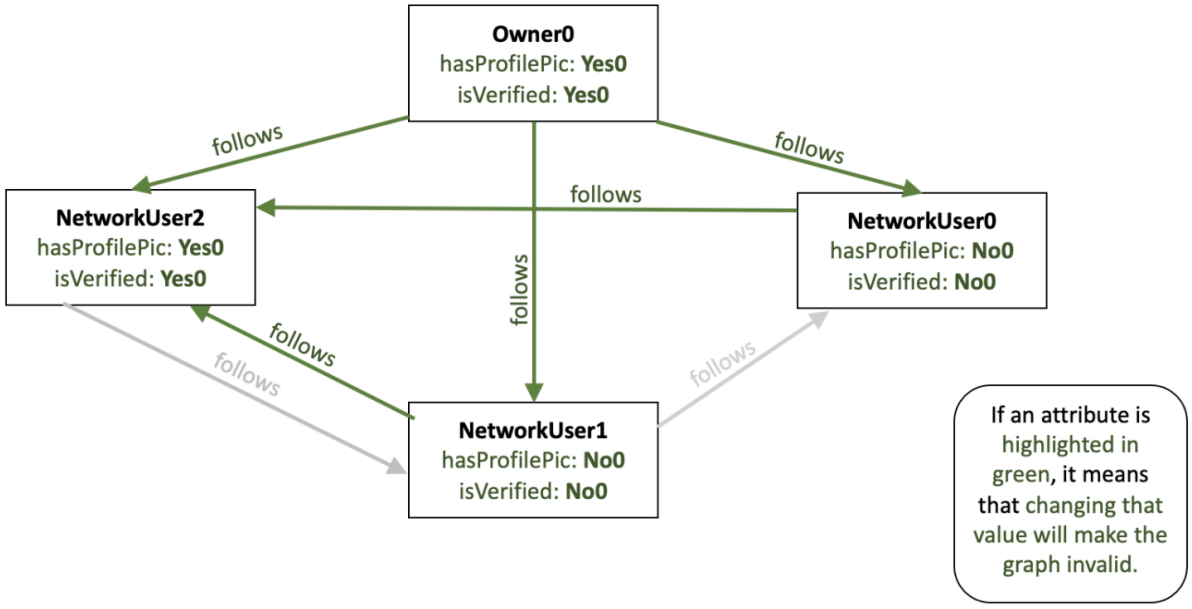
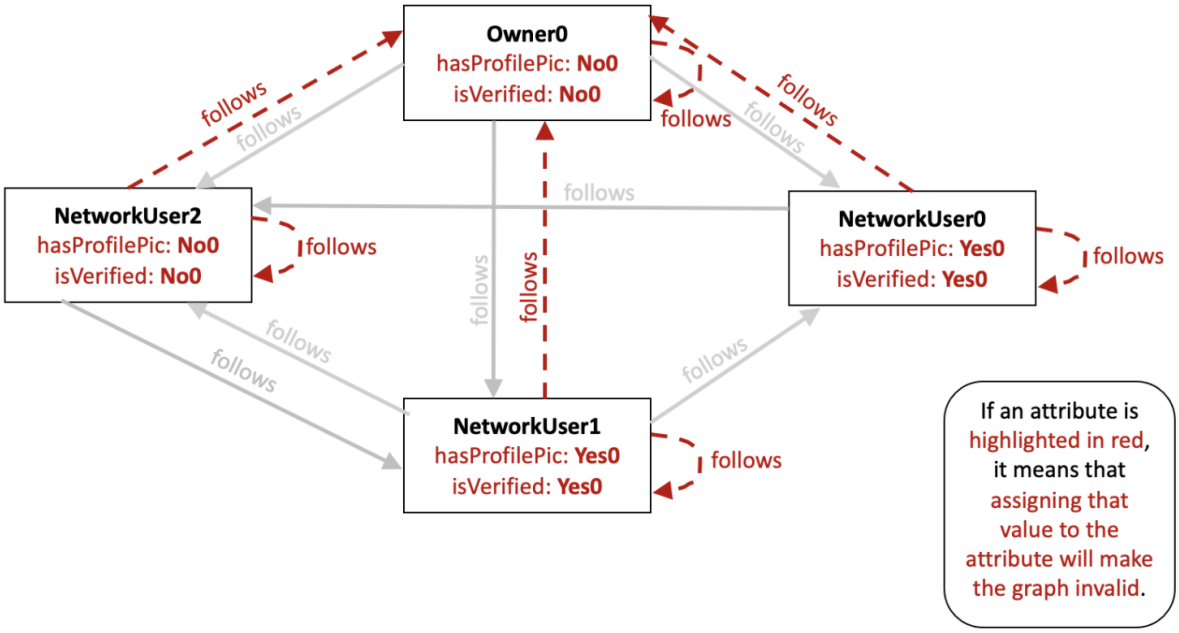


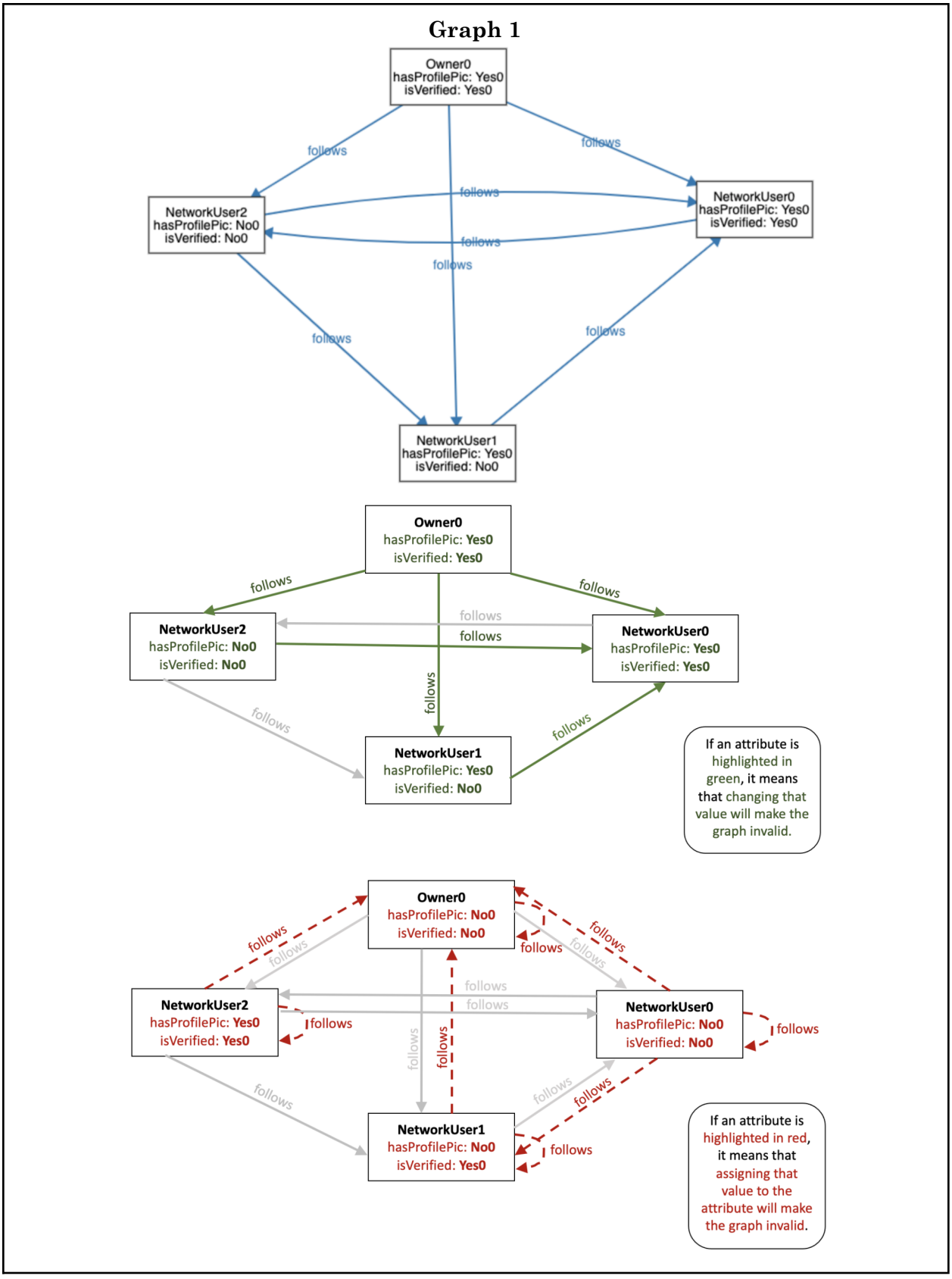
Figure 34. Updated Negative Local Necessity Description for Experimental Group

Here we see `graphA` with some red-dotted lines. The red-dotted lines mean that if you tried to add any red-dotted line, `graphA` would become invalid. As an example, if we were to add the red-dotted "`follows`" line from `NetworkUser1` to `Owner0`, `graphA` would become invalid. Adding the red-dotted lines makes `graphA` invalid because there are other things in `graphA` that depend on that red-dotted line NOT being there. For the attributes of `NetworkUsers` and `Owners` (`isVerified` and `hasProfilePic`), if the attribute is highlighted in read, it means that assigning that value to the attribute will make the graph invalid. For example, `Owner0`'s "`hasProfilePic`" field is "`No0`" highlighted in red. This means that making `Owner0`'s "`hasProfilePic`" field "`No0`" makes the graph invalid because there are other things in `graphA` that depend on `Owner0`'s "`hasProfilePic`" field to not be "`No0`". A gray line represents existing facts in `graphA`.

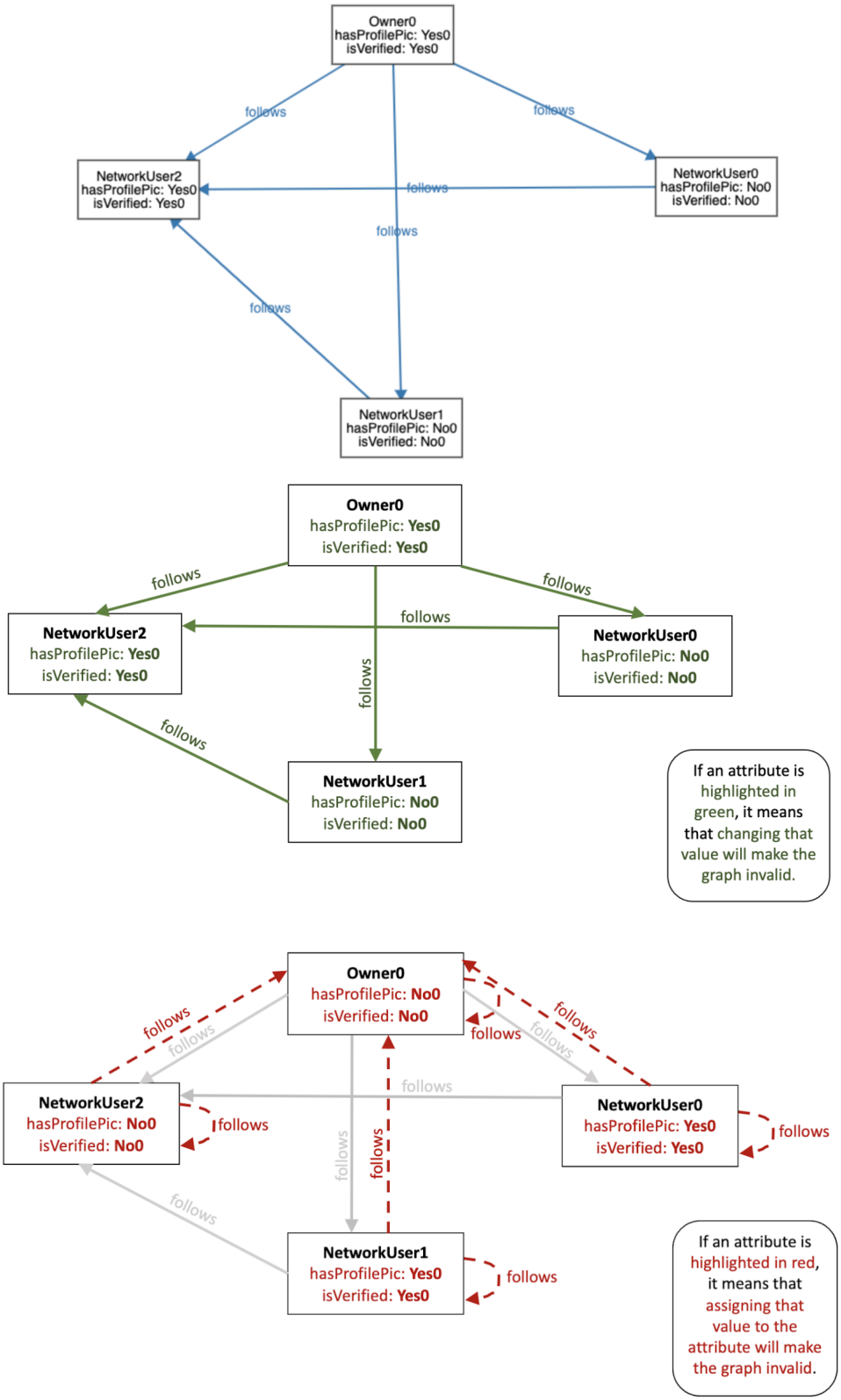


The four graphs shown to the participants of the updated experimental group can be seen in figure 35.

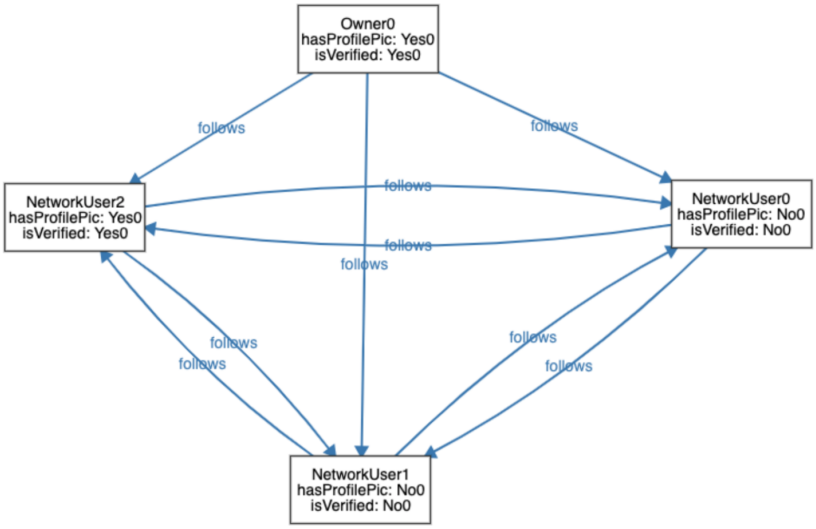
Figure 35. Updated Graphs for the Experimental Group

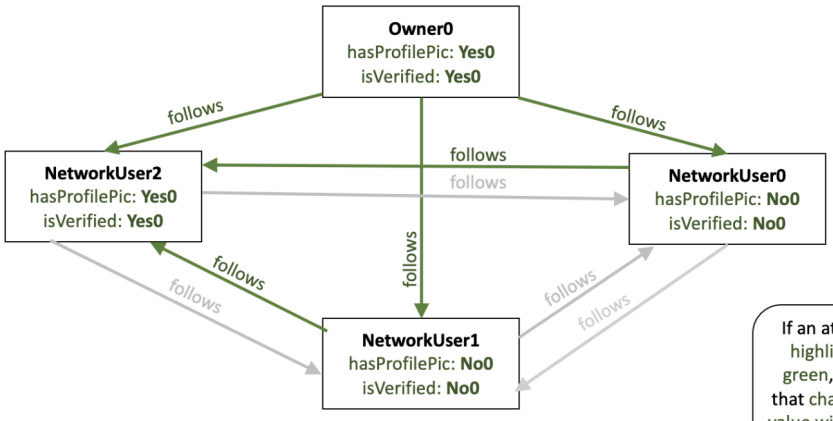


Graph 2

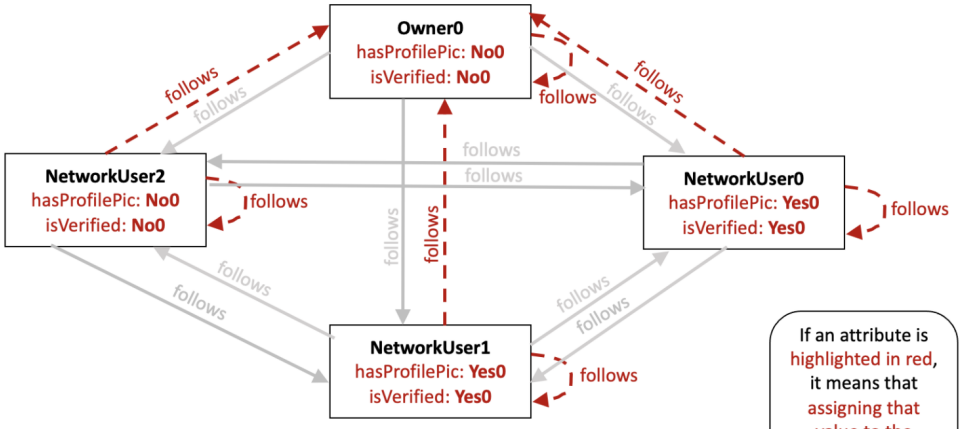

Owner0
hasProfilePic: Yes0
isVerified: Yes0
follows
follows
NetworkUser2
hasProfilePic: Yes0
isVerified: Yes0
follows
NetworkUser0
hasProfilePic: No0
isVerified: No0
follows
follows
NetworkUser1
hasProfilePic: No0
isVerified: No0
Owner0
hasProfilePic: Yes0
isVerified: Yes0
follows
follows
NetworkUser2
hasProfilePic: Yes0
isVerified: Yes0
follows
NetworkUser0
hasProfilePic: No0
isVerified: No0
follows
follows
NetworkUser1
hasProfilePic: No0
isVerified: No0
If an attribute is highlighted in green, it means that changing that value will make the graph invalid.
Owner0
hasProfilePic: No0
isVerified: No0
follows
follows
follows
follows
follows
NetworkUser2
hasProfilePic: No0
isVerified: No0
follows
follows
follows
NetworkUser0
hasProfilePic: Yes0
isVerified: Yes0
follows
follows
follows
NetworkUser1
hasProfilePic: Yes0
isVerified: Yes0
follows
If an attribute is highlighted in red, it means that assigning that value to the attribute will make the graph invalid.

Graph 3



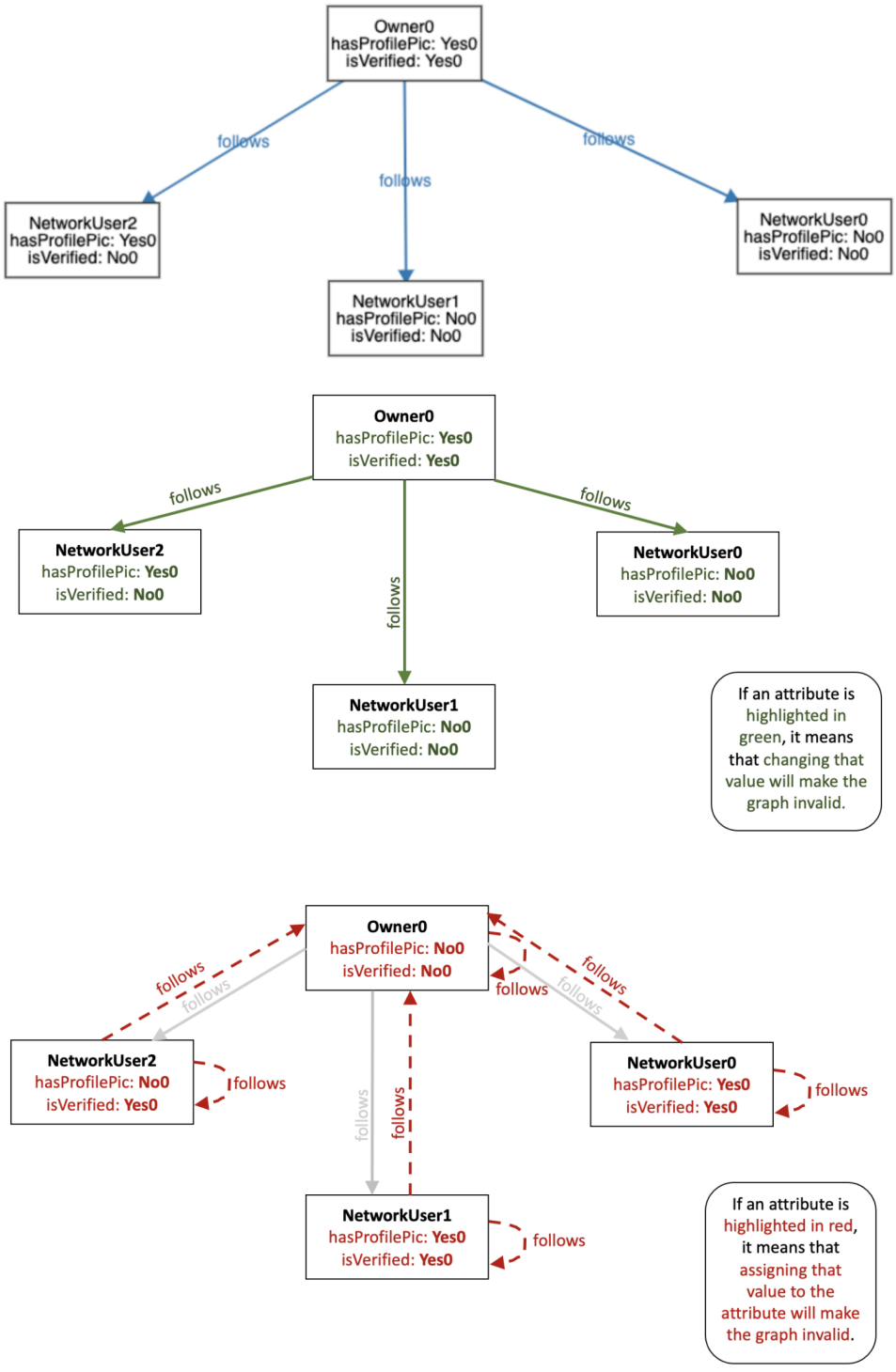


If an attribute is highlighted in green, it means that changing that value will make the graph invalid.



If an attribute is highlighted in red, it means that assigning that value to the attribute will make the graph invalid.

Graph 4


Owner0
hasProfilePic: Yes0
isVerified: Yes0
follows
follows
follows
NetworkUser2
hasProfilePic: Yes0
isVerified: No0
NetworkUser1
hasProfilePic: No0
isVerified: No0
NetworkUser0
hasProfilePic: No0
isVerified: No0
Owner0
hasProfilePic: Yes0
isVerified: Yes0
follows
follows
follows
NetworkUser2
hasProfilePic: Yes0
isVerified: No0
NetworkUser1
hasProfilePic: No0
isVerified: No0
NetworkUser0
hasProfilePic: No0
isVerified: No0
If an attribute is highlighted in green, it means that changing that value will make the graph invalid.
Owner0
hasProfilePic: No0
isVerified: No0
follows
follows
follows
follows
follows
follows
follows
NetworkUser2
hasProfilePic: No0
isVerified: Yes0
NetworkUser0
hasProfilePic: Yes0
isVerified: Yes0
NetworkUser1
hasProfilePic: Yes0
isVerified: Yes0
If an attribute is highlighted in red, it means that assigning that value to the attribute will make the graph invalid.

Results

As in the first version of studies for Mechanical Turk, in order to analyze my results I filtered out the responses that had more than one incorrect answer for the initial questionnaire. This resulted in 22 valid responses for the updated experimental group. The distributions for correctly classifying each rule for the control, original experiment, and updated experimental groups can be seen in figures 36-37.

Figure 36. Results for Control, Experimental, and Updated Experimental Group Rules 1-4.

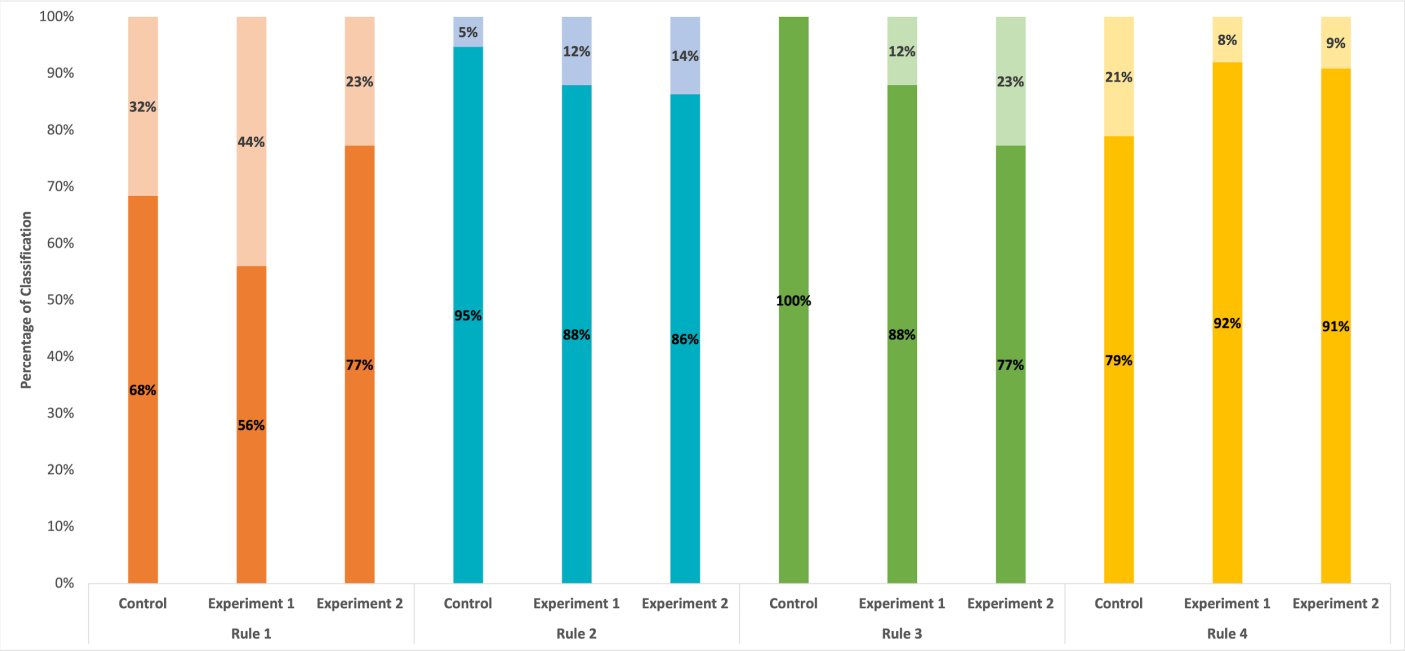
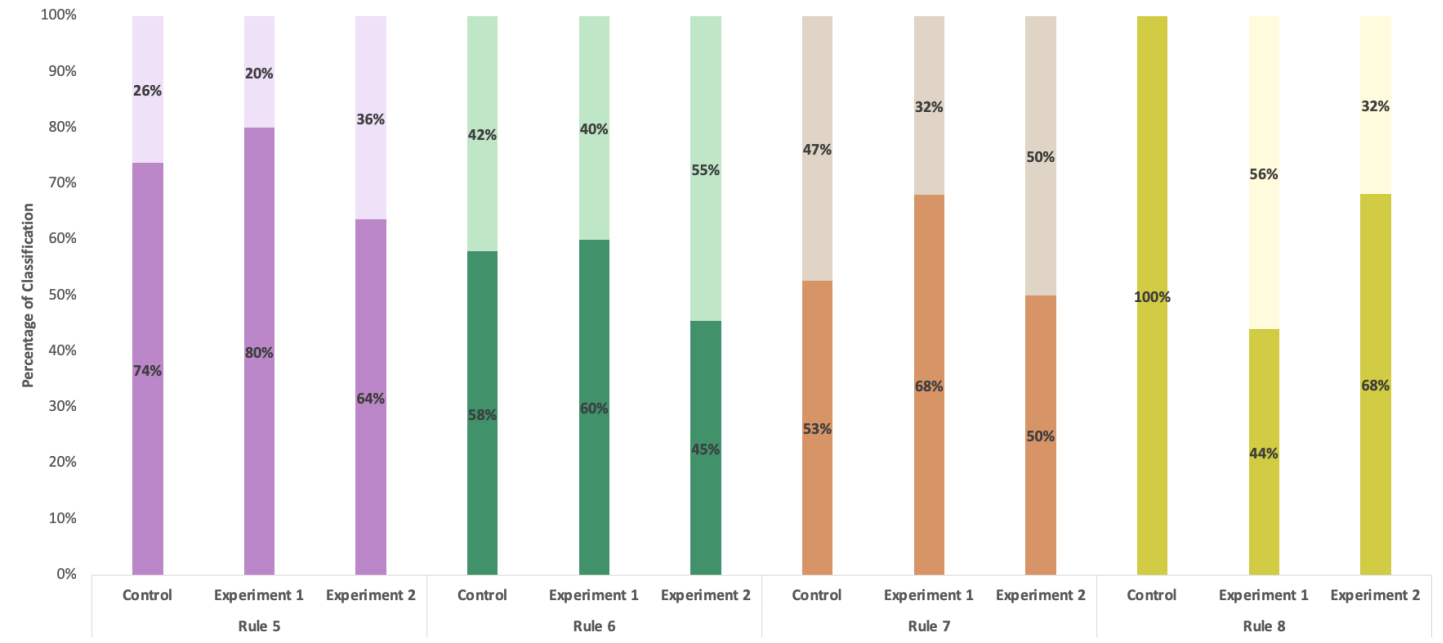


Figure 37. Results for Control, Experimental, and Updated Experimental Group Rules 4-8.



From the previous images, it can be noted that the updated experimental group did better than the original experimental group in the classification of only two rules: rule number one and rule number eight. Of these two rules, the updated experimental group did better than the control group in the classification of rule 1. In order to measure the impact of changing the way that the graphs looked, I performed the same Chi-squared test as the one in section 5.2 but between the original experimental group and the updated experimental group, and between the control group and the updated experimental group. I also calculated Hedges' G for these same group combinations.

The results for calculating the Chi-Squared distribution for each rule for the original experimental group and the updated experimental group with a significance level of 0.05 can be seen in figure 38.

Figure 38. Chi-Squared Analysis Results

Rule Number	Chi-Squared Result	Calculated P-value	Conclusion
1	2.358442082	0.124606575	Results are not statistically significant
2	0.028137472	0.866786033	Results are not statistically significant
3	0.953420746	0.328850102	Results are not statistically significant
4	0.017885835	0.893609807	Results are not statistically significant
5	1.566022213	0.210785626	Results are not statistically significant
6	0.994380165	0.318674175	Results are not statistically significant
7	1.574323308	0.209580274	Results are not statistically significant
8	2.768508159	0.096135722	Results are not statistically significant

Given that no rule was statistically significant, I could not conclude that there was a relationship between the type of local necessity graph that participants observed and their performance in correctly classifying the rules of the model.

I then continued to calculate Hedges' G for both experimental groups. I once again calculated a score for each participant as done in section 5.2.

Figure 39. Average Scores and Standard Deviations for Original and Updated Experimental Groups

Original Experimental Group	Updated Experimental Group
Average Score: 5.76 Standard Deviation: 1.42243922	Average Score: 5.590909091 Standard Deviation: 1.708775415
Hedges' G: 0.108211 **Small Effect Size**	

For the analysis made between the control group and updated experimental groups, my Chi-Squared distribution for each rule with a significance level of 0.05 can be seen below.

Figure 40. Chi-Squared Analysis Results

Rule Number	Chi-Squared Result	Calculated P-value	Conclusion
1	0.406908801	0.52354266	Results are not statistically significant
2	0.811861503	0.367570469	Results are not statistically significant
3	**4.917929293**	**0.026579356**	**Results are statistically significant**
4	1.167691957	0.279875989	Results are not statistically significant
5	0.475340449	0.490540517	Results are not statistically significant
6	0.63148781	0.4268101	Results are not statistically significant
7	0.028258145	0.866503357	Results are not statistically significant
8	**7.290106952**	**0.006933535**	**Results are statistically significant**

For these results, it is interesting to note that the classification of the rules 3 and 8 were statistically significant favoring the control group. While the significance in the classification of rule 8 also occurred with the original experimental group, the significance for the classification of rule 3 is new. Rule 3 is "The `Owner` needs to follow all `NetworkUsers`". It is interesting that statistical significance for these studies have come from the 2 properties related to the `Owner` instead of the other `NetworkUsers`. These results might signal towards the fact that there is still room for improvement for the display of local necessity in Sterling, as participants might be getting confused by the meanings of the positive and negative local necessity graphs. I also think the aforementioned because the updated experimental group only did better than the original experiment group in 2 questions: 1 and 8; therefore, even though there was no statistical significance for question 1, there might be some confusion still on how local necessity works.

I then continued to calculate Hedges' G for the control and updated experimental groups. I once again calculated a score for each participant as done in section 5.2. Results can be seen in figure 41.

Figure 41. Average Scores and Standard Deviations for Control Group and Updated Experimental Group

Control Group	Updated Experimental Group
Average Score: 6.263157895 Standard Deviation: 1.240165995	Average Score: 5.590909091 Standard Deviation: 1.708775415
Hedges' G: 0.445001 **Small Effect Size**	

6. Analysis

While the results from the study ran in CSCI 1710 do not allow me to define any findings regarding the usefulness of local necessity, they do allow me to make a comparison between my results and those of Danas et al. (2017). In my study, I exposed students to having local necessity and not having local necessity, while Danas et al. (2017) exposed students to Amalgam's provenances. In my study, from the logs that I found, 8/8 students exposed to local necessity and 29/29 students exposed to nothing chose the correct fix to add `KittyBacon` to the `FancyFelineFoundation`. These results are more favorable than the results found in Danas et al. (2017) where one fifth of the students that were exposed to provenances made a fix to add `KittyBacon`to the `CoolCatClub` that went against other constraints in the model, therefore making the model invalid. This sheds some light on the usability of provenances itself. Are provenances useful? Or are they hurting participant's understanding of their model and over-constraints instead of helping them?

Students performance in my studies might have also been impacted by the presence of the satisfying instance at the beginning of the assignment. Therefore, the differences in the results between my studies and Danas et. al (2017) could be attributed to either provenances being confusing or the initial satisfying instance being very helpful for users to debug their models. With this in mind, there is an opportunity to pursue a future study that compares these three options directly in more controlled environments. In the case that the culprit was the image from a satisfying instance of the model, that could be a significant finding and could shape the way that assignments are approached for the course and other formal methods language assignments.

In regards to the results from the first Mechanical Turk study, there was only one significant case that favored the control group. This points towards the fact that it might be more useful for users to observe multiple instances in Sterling to identify constraints rather than fewer instances with their local necessity. This potentially contradicts the statement made in Nelson et al. (2017) saying that Amalgam is able to point out under-constraints and over-constraints in a clearer way than observing multiple instances and trying to figure out the rules that govern the model that generated them. This is a significant finding that gives me insight on the question of whether or not Amalgam's local necessity is useful in helping users understand their model better. Additionally, as mentioned in section 5.2, the statistical result for this study suggested that some ways of displaying local necessity might be confusing to non-expert users, especifically differentiating between positive and local necessity. This led to the second round of Mechanical Turk studies focused on testing alternative visualizations of local necessity.

The results from the second Mechanical Turk study point towards the fact that there wasn't a significant difference between the original display of local necessity and the modified display of local necessity. However, I believe that this still shows the need to iterate on

better visualizations for local necessity, as local necessity might still be a confusing subject to grasp for new users. Additionally, when comparing the results of the updated experimental group with the control group, there were 2 statistically significant results favoring the control group. This once again clarifies that there is no significant benefit in utilizing local necessity versus exposing users to multiple instances of a given model; however, it emphasizes that even the updated visualization of local necessity can still be harmful to non-expert participants.

Additionally, the analysis made of the benefits and disadvantages of implementing Amalgam in Forge is meant to act as a guide for future developers seeking to introduce a tool to this language. Hopefully, the benefits and limitations listed will allow developers to determine whether Forge is the language that would best fit their tool, and will also give Forge developers the opportunity to understand the strengths and weaknesses of the language for future improvement.

7. Conclusion

In conclusion, my honors project approached the question of whether or not local necessity was useful in aiding a user's debugging process and understanding of their models. Through multiple user studies, both to test the direct usability of the tool and the usefulness of local necessity graphs versus regular graphs, I was able to determine that there is no significant benefit in exposing users to a models' instances with local necessity than exposing users to multiple instances to increase their understanding of a model's constraints. I was also able to determine that there are some cases where the visualization of local necessity can be harmful to non-expert users. In addition, the results shed some light on the usefulness of provenances in Danas et al. (2017) or the presence of the satisfying instance at the beginning of an assignment, and encourages future research on the matter.

Finally, this paper made an analysis of implementing Amalgam in the programming language, Forge. In this thesis, I outlined multiple benefits and disadvantages of implementing such a tool in this language, which is meant to act as a guide for future developers. Additionally, I outlined a set of suggestions to improve developers' experiences when working in Forge, which can also be used by Forge developers to improve the language.

7.1 Limitations

When it comes to my first run of user studies, the main limitation was that I was not able to determine the usability of local necessity given the failure of logging. Because of this, I was not able to answer whether or not local necessity is useful for users during their debugging processes.

For the studies run in Mechanical Turk, there exists the limitation that the studies are vulnerable to the specific visualizations, domain, and wording chosen for the questions. For once, the translations made from the constraints in the model in figure 18 into English might have been unclear to some users. Additionally, as noted in the analysis section, there might still exist some confusion on local necessity graphs. While this research paper looked into an alternative visualization, results pointed towards there being a greater opportunity for improvement in that area. Finally, another limitation is that the user studies that I ran on Mechanical Turk were only run for a medium number of people (25-27), which is not a sample size big enough to develop strong statistical analysis.

8. Future Improvements

(1) **User Study for Usability:** Given that I was not able to make a direct analysis of the usability of the tool, there is room for running the same study once again next year in CSCI 1710 and logging student's actions.

(2) **Improve Performance of Amalgam in Forge:** In order to make Amalgam faster and therefore more useful for users, Forge developers might need to work on another way of evaluating an instance without invoking the solver directly. Invoking the solver takes up a lot of time, and therefore slows down the tool a lot.

(3) **Test Usability of Provenances:** Given that the results from the Mechanical Turk studies shed some light on whether or not provenances were helpful to students in Danas et al. (2017), a study focused on comparing provenances, local necessity, and being exposed to simple Forge can be extremely useful.

(4) **Test Usability of Initial Satisfying Instance:** Since the positive results for the laboratory might also be attributed to the presence of the initial satisfying instance of the model that students needed to complete, there is an opportunity to develop a study focused on this directly and determine the benefits of having the initial instance.

(5) **Improving Visualization of Local Necessity:** The study results show that there is still a lot of room for improvement for the display of local necessity in Sterling. A more detailed analysis could be done where multiple Mechanical Turk studies are run exposing participants to different graph formats. This would also increase the usability of the tool.

(6) **Running Talk-Aloud Studies:** Another way of getting some insight into the usability of local necessity and provenances could be by running talk-aloud studies. In these studies, participants could be exposed to multiple models in which they are given the opportunity of using Amalgam in some but not in others. Having access to participant's thoughts while using the tool can be very beneficial in improving it and understanding its usability better.

9. Acknowledgements

I would like to thank Professor Tim Nelson for his unconditional support, and Professor Shriram Krishnamurthi for all of his support and guidance throughout this process. I would also like to thank Abigail Siegel for all of her support and help implementing Amalgam. Thank you to my friends and family for their words of encouragement.

10. References

[1] Lahtinen, E., Ala-Mutka, K., & Järvinen, H. (2005). A study of the difficulties of novice programmers. *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education - ITiCSE '05*. doi:10.1145/1067445.1067453

[2] Loksa, D., & Ko, A. J. (2016). The role of self-regulation in programming problem solving process and success. *Proceedings of the 2016 ACM Conference on International Computing Education Research*. doi:10.1145/2960310.2960334

[3] Prather, J., Pettit, R., McMurry, K., Peters, A., Homer, J., & Cohen, M. (2018). Metacognitive difficulties faced by novice programmers in automated assessment tools. *Proceedings of the 2018 ACM Conference on International Computing Education Research*. doi:10.1145/3230977.3230981

[4] Garner, S., Haden, P., & Robins, A. (2005). My Program is Correct But it Doesn't Run: A Preliminary Investigation of Novice Programmers' Problems. *In Proc. Seventh Australasian Computing Education Conference (ACE2005).*

[5] Jackson, D. (2016). *Software abstractions: Logic, language, and analysis*. Cambridge, MA: The MIT Press.

[6] Nelson, T., Danas, N., Dougherty, D. J., & Krishnamurthi, S. (2017). The power of "why" and "why not": Enriching scenario exploration with provenance. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. doi:10.1145/3106237.3106272

[7] Larch languages and tools for formal specification [Preface]. (1993). In 1251893671 927138261 J. V. Guttag & 1251893672 927138261 J. J. Horning (Authors), *Larch languages and tools for formal specification* (pp. 4-5). New York: Springer.

[8] Porncharoenwase, S., Nelson, T., & Krishnamurthi, S. (2018). Composat: Specification-guided coverage for model finding. *Formal Methods,* 568-587. doi:10.1007/978-3-319-95582-7_34

[9] Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., & Dougherty, D. J. (2017). User studies of Principled Model Finder Output. *Software Engineering and Formal Methods,* 168-184. doi:10.1007/978-3-319-66197-1_11

[10] Wrenn, J., & Krishnamurthi, S. (2020). Will students write tests early without coercion? *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. doi:10.1145/3428029.3428060

[11] Denny, P., Prather, J., Becker, B. A., Albrecht, Z., Loksa, D., & Pettit, R. (2019). A closer look at metacognitive scaffolding. *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*. doi:10.1145/3364510.3366170

[12] Leino, K. Rustan M. (2010). Dafny: An automatic Program verifier for functional correctness. *Logic for Programming, Artificial Intelligence, and Reasoning,* 348-370. doi:10.1007/978-3-642-17511-4_20

[13] Leino, K. Rustan M. (2010). Dafny: An automatic Program verifier for functional correctness. *Logic for Programming, Artificial Intelligence, and Reasoning,* 348-370. doi:10.1007/978-3-642-17511-4_20

[14] Mason, W., Suri, S.: Conducting behavioral research on Amazon's Mechanical Turk. Behavior research methods 44(1), 1–23 (2012)

[15] Dyer, A. T. (2020). *Lightweight formal methods in scientific computing* (Unpublished master's thesis). North Carolina State University.

[16] Dyer, T. (2020). Sterling. Retrieved March 23, 2021, from https://sterling-js.github.io/

[17] Bornholt, J., & Torlak, E. (2017). Synthesizing memory models from framework sketches and litmus tests. *ACM SIGPLAN Notices, 52*(6), 467-481. doi:10.1145/3140587.3062353

11. Appendix

Control Group Handout

Forge Lab 1: Graphs

Monday, February 1, and Tuesday, February 2

You should only be completing this lab if you are in the Monday 4-6 pm and Tuesday 8-10 pm sections. If you are not a part of these sections, please return to the course's website and pick the correct handout.

Overview

Overview of Lab: Throughout this lab, we will be modeling a set of `Cat`s, where each `Cat` has a set of `friends`. Among other properties, friendship is symmetric, meaning that if CatA is friends with CatB, CatB is also friends with CatA. We will create a specific `Cat` named `KittyBacon`, whose connections are all of the other `Cat`s (a relation named "`connectionsOf`"). `KittyBacon` will create a foundation named "`FancyFelineFoundation`" `(FFF)` where the `membersOfFoundation` are `KittyBacon`'s connections (all of the other `Cat`s).

To give you an idea of what your solution should look like, below is an output instance from a **correct implementation of this lab (the TA's solution)**. Take some time to understand how the various relations work in the model and feel free to use this instance as reference throughout the lab. The following instance contains 3 `Cat`s (`Cat0, Cat1, KittyBacon0`). `FFF0` represents the `FancyFelineFoundation`.

Graph View:

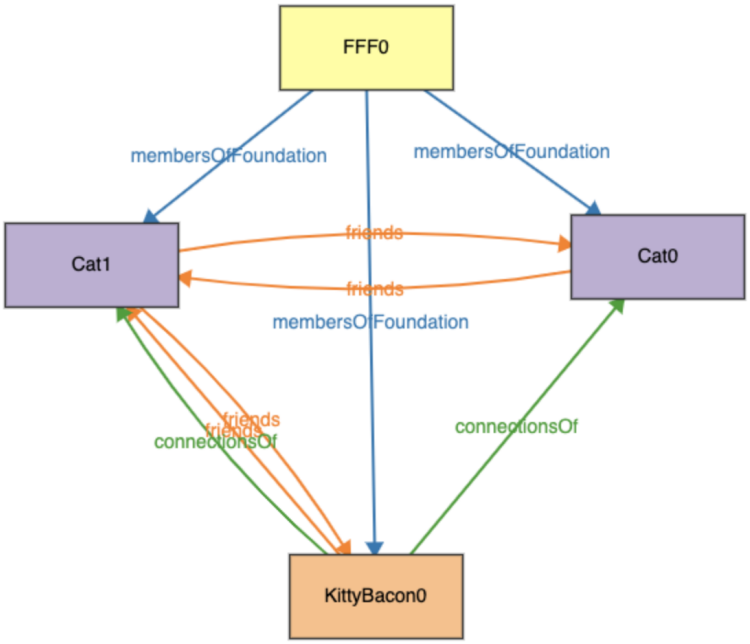


Table View:

Cat

Cat0
Cat1

KittyBacon

KittyBacon0

FFF

FFF0

univ ◀ friends

univ	univ
Cat0	Cat1
Cat1	Cat0
Cat1	KittyBacon0
KittyBacon0	Cat1

univ ◀ connectionsOf

univ	univ
KittyBacon0	Cat0
KittyBacon0	Cat1

univ ◀ membersOfFoundation

univ	univ
FFF0	Cat0
FFF0	Cat1
FFF0	KittyBacon0

Setup

Start by opening a new file in DrRacket. At the very top of the file, make sure to include the following line, where <email> is your Gradescope email. If there is already a `#lang` line, you should replace it with this one.

```
#lang forge "lab1" "<email>"
```

For example, one of our TA's lab looks as follows:

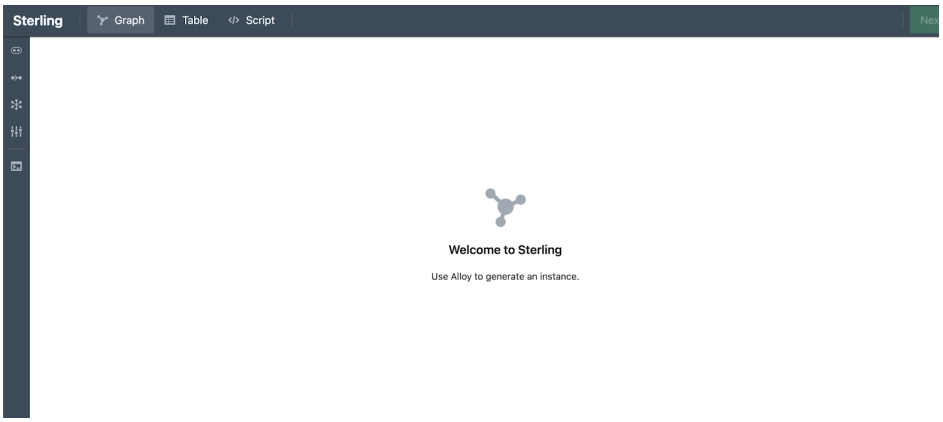
```
#lang forge "lab1" "x0f00iexvy4yoidl@gmail.com"
```

You might see a message at the bottom of your Forge screen saying “uncaught exn: "Email verification failed." or “call-with-output-file: forbidden (write delete) access to …”. Feel free to ignore these.

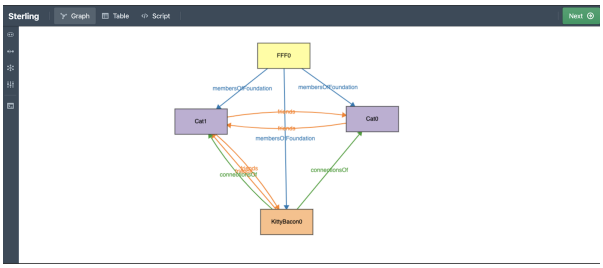
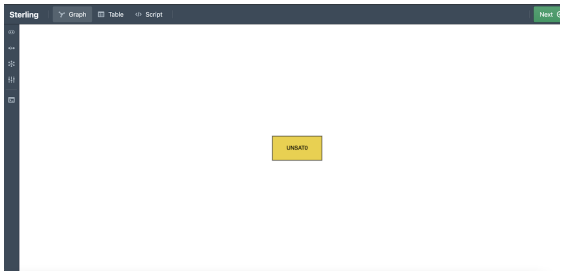
Afterwards, run your code. You might be prompted to write your email address and the name of the assignment once again. If you are, write the same email as above (without quotations) and in the assignment name write lab1 (without quotations as well).

Note on Running Your Code

Whenever you run your code, a window like the one below will appear in your Internet browser. This program is called Sterling.



After waiting a few seconds, a graph will show up in that window if the spec is satisfiable. In the case that the spec is unsatisfiable, a box saying `UNSAT0` will show up.

Satisfiable Spec	Unsatisfiable Spec
	

We Are All Set!

Begin with the following Forge signature describing `Cat`s, each of which has a set of `friends`:

```
sig Cat {
  friends : set Cat
```

```
}
run {}
```

As crucial as the friendship of kittens is, we're using it here as a playful framing of very real problems in computer science.

Averting Catastrophe

Run this Forge model and view several instances in the visualizer by clicking *Next*. This model has a few unpleasant characteristics that we would like to constrain.

No Lonely Kitten

Make sure that all Cats have at least one friend. Using the predicate below as a template, replace `...` with your implementation:

```
pred NoLonelyKittens {
  …
}
```

Cats are Friends with *Other* Kittens

Some of the cats are claiming themselves as friends! Introduce a fact ensuring that the friends of a cat do not include itself. Using the predicate below as a template, replace `...` with your implementation:

```
pred OutsideFriends {
  …
}
```

Friendship Isn't A One-Way Street

Well, *Friendship* might be a one-way street in Providence, but these cats aren't from around here. Visualize an instance of your model so far, and note that the arrows connecting instances of `Cat` may be one-directional. Introduce a predicate `TwoWayStreet` ensuring that if one cat considers another cat as a friend, the other cat reciprocates the friendship:

```
pred TwoWayStreet {
  …
}
```

You can combine these all together into one predicate:

```
pred friendship {
      NoLonelyKittens
      OutsideFriends
      TwoWayStreet
}
```

Try running this predicate with `run friendship` to see all the cats! It is normal for the first instance to be empty, try clicking "Next" to see other instances!

Kitty Bacon

There's one special `Cat` in our universe, an actor by the name of Kitty Bacon! Add this signature to your model:

```
one sig KittyBacon extends Cat { connectionsOf : set Cat }
```

If you view an instance of this model, you should see exactly one `Cat` with the name `KittyBacon`.

Degrees of Kitty Bacon

Kitty Bacon is very connected. All kitties are in his connections. Add the following predicate:

```
pred ConnectedKittyBacon {
    --connectionsOfKittyBacon will be defined soon!
    connectionsOfKittyBacon
    Cat - KittyBacon = KittyBacon.connectionsOf
}
```

`ConnectedKittyBacon` depends on the relation `KittyBacon.connectionsOf`, which should consist of the union of `KittyBacon`'s `friends` of `friends` (not including `KittyBacon` themself) and `KittyBacon`'s `friends`.

It turns out that doing `KittyBacon.friends.friends` gets you `KittyBacon`'s `friends` of `friends`! Play around with this in the evaluator to see why that is. With that in mind, complete the definition below. **Make sure to write this predicate above** `ConnectedKittyBacon`:

```
pred connectionsOfKittyBacon {
    friendship
    KittyBacon.connectionsOf = ...
}
run {ConnectedKittyBacon and connectionsOfKittyBacon}
```

Look at the visualization of your instance. Are *all* cats connected to `KittyBacon` via their network of friends? If you got `UNSAT`, double-check that your definition of `KittyBacon.connectionsOf` isn't including too many cats.

A common pattern in Forge is to test a suspicion by phrasing it differently in a second predicate, and checking if both predicates are equivalent. An alternative way to express connected-via-friends is the *transitive closure operator*, `^`.

`KittyBacon.^friends` includes `friends`, `friends` of `friends`, `friends` of `friends` of `friends`, and so on. Include the following predicate in your spec:

```
pred SuperConnected {
    connectionsOfKittyBacon
    Cat - KittyBacon in KittyBacon.^friends
}
```

Two predicates are equivalent (up to bound only) if and only if (`iff`) one implies the other. Fill in the predicate below:

```
pred ConnectedKittyBacon_equals_SuperConnected {
  …
}
```

Finally, check the predicate:

```
check ConnectedKittyBacon_equals_SuperConnected for exactly 3 Cat
```

If you run this `check` and a graph pops-up in the visualizer, it means that you got a counter-example. If the visualizer shows an `UNSAT0` box, then your `check` passed. Were any counter-examples found?

What happens if you increase the scope of the model to include an additional `Cat`?

```
check ConnectedKittyBacon_equals_SuperConnected for exactly 4 Cat
```

Modify KittyBacon.`connectionsOf` to include their `friends` of `friends` of `friends` and check for counterexamples again. If none are found, increase the scope of the `check` to `exactly 5 Cat`. Is it possible to modify KittyBacon.`connectionsOf` using only the `+` operator so that for any number of degrees of separation, and for any number of cats, no counterexample to this predicate can be found? If not, why? Write your response in a comment.

The Fancy Feline Foundation

Because `KittyBacon` is so fancy, he would like to create a foundation with all of his connections. Add the following signature to your model:

```
one sig FFF  { membersOfFoundation: set Cat }
```

All of `KittyBacon`'s connections are part of the `FancyFelineFoundation`:

```
pred FancyFelineFoundation {
    ConnectedKittyBacon
```

```
    FFF.membersOfFoundation = KittyBacon.connectionsOf
}
```

Since `KittyBacon` is such a fancy feline (he started the `FancyFelineFoundation`, after all), we can check whether `KittyBacon` is part of the `FancyFelineFoundation`:

```
pred KittyBaconIsAFancyFeline {
    FancyFelineFoundation implies KittyBacon in FFF.membersOfFoundation
}
```

Now, add the following check:

```
check KittyBaconIsAFancyFeline for exactly 3 Cat
```

What do you see when you run your check? Do you get any counterexamples? Is `KittyBacon` ever a part of the `FancyFelineFoundation`?

If `KittyBacon` is never a part of the `FancyFelineFoundation`, make sure to modify your code to include him (after all, he started it!).

After your fix, run your `check` once again and see if there are any counterexamples. If there are none (meaning that no graphs show up), **please write what your fix was in a comment and your reasoning behind it.**

Now, we are going to add a final predicate to see a graph representing our entire model (including `KittyBacon` in the `FancyFelineFoundation`). Please add the following predicate:

```
pred seeAllCats {
    FancyFelineFoundation
    KittyBacon in FFF.membersOfFoundation
}
```

You might be wondering why we are adding this. Well, the fix that we added might have solved the problem of `KittyBacon` not being a part of the `FancyFelineFoundation`, but **did our fix lead to any unintended consequences?** This is very common in programming! While normally we would also write a few example tests, in the scope of this lab we are just going to run the predicate and see a couple of instances to make sure that we did not break any properties that should hold.

Include the following run statement in your spec:

```
run seeAllCats for exactly 3 Cat
```

Make sure to keep the visualizer window open from your run since you will need to show it to the TAs before check-off.

Check-Off

Call a TA over to talk about your responses.

Experimental Group Handout

Forge Lab 1: Graphs

Monday, February 1, and Tuesday, February 2

You should only be completing this lab if you are in the Monday 8-10 am and Tuesday 4-6 pm sections. If you are not a part of these sections, please return to the course's website and pick the correct handout.

Overview

Overview of Lab: Throughout this lab, we will be modeling a set of `Cat`s, where each `Cat` has a set of `friends`. Among other properties, friendship is symmetric, meaning that if CatA is friends with CatB, CatB is also friends with CatA. We will create a specific `Cat` named `KittyBacon`, whose connections are all of the other `Cat`s (a relation named "`connectionsOf`"). `KittyBacon` will create a foundation named "`FancyFelineFoundation` `(FFF)` where the `membersOfFoundation` are `KittyBacon`'s connections (all of the other `Cat`s).

To give you an idea of what your solution should look like, you will be given an output instance from a **correct implementation of this lab (the TA's solution)**. In addition, you will be able to see what is and isn't **locally necessary** for that specific instance, a concept that Tim went over in Friday's lecture. For an overview of Local Necessity, check out the following section.

What is Local Necessity?

Local Necessity tells you about what facts need to hold in order for an instance to satisfy the spec, in the context of the rest of the instance. There are two types of local necessity: positive and negative.

(1) **Positive local necessity (LN+)** are those facts that are true in the instance, and cannot be flipped to false without violating the spec or making other changes to the instance.
(2) **Negative local necessity (LN-)** are those facts that are false in the instance, and cannot be flipped to true without violating the spec or making other changes to the instance.

As a brief example, consider a spec that models registering courses at Brown that does not allow taking courses simultaneously and requires students to register for all components of a given course.

- Positive example: Imagine that CSCI 1710 requires a mandatory section. If an instance of your model were to show 1710 as a registered course, it is **locally necessary** for the section **to also be** registered (given that 1710 is registered, the section must also be registered or the spec is violated).
- Negative example: Imagine that you are shopping CSCI 1710 and APMA 1160 (both MWF 10-10:50 am.) If your instance shows CSCI 1710 as a registered course, **it is**

locally necessary for APMA 1160 to **not** be in your list of registered courses (given that you are registered for 1710, you cannot also register for APMA 1160 without violating the spec).

If you have any questions about local necessity, please call a TA over.

An output instance of the TA's solution (in graph and table view) and its local necessity can be seen below. Take some time to understand how the various relations work in the model and feel free to use this instance as a reference throughout the lab. You are also encouraged to analyze your own instances' local necessity. The following instance contains 3 `Cat`s (`Cat0, Cat1, KittyBacon0`). `FFF0` represents the `FancyFelineFoundation`.

Graph View:

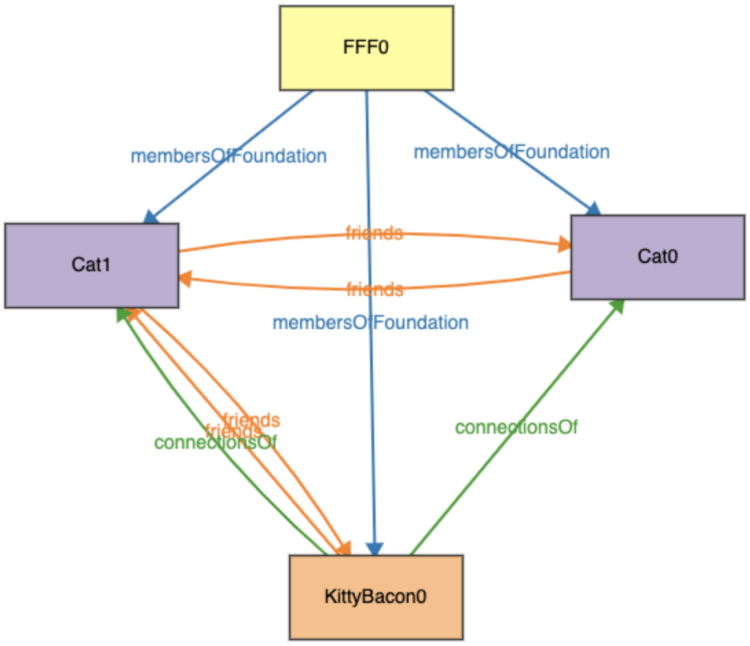


Table View:

Cat

Cat
Cat0
Cat1

KittyBacon

KittyBacon
KittyBacon0

FFF

FFF
FFF0

univ ◀ friends

univ	univ
Cat0	Cat1
Cat1	Cat0
Cat1	KittyBacon0
KittyBacon0	Cat1

univ ◀ connectionsOf

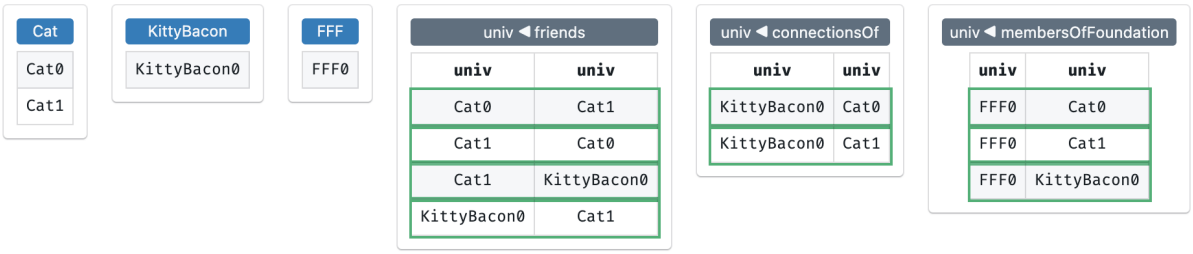
univ	univ
KittyBacon0	Cat0
KittyBacon0	Cat1

univ ◀ membersOfFoundation

univ	univ
FFF0	Cat0
FFF0	Cat1
FFF0	KittyBacon0

Local Necessity:

Positive Local Necessity:



Cat

Cat
Cat0
Cat1

KittyBacon

KittyBacon
KittyBacon0

FFF

FFF
FFF0

univ ◀ friends

univ	univ
Cat0	Cat1
Cat1	Cat0
Cat1	KittyBacon0
KittyBacon0	Cat1

univ ◀ connectionsOf

univ	univ
KittyBacon0	Cat0
KittyBacon0	Cat1

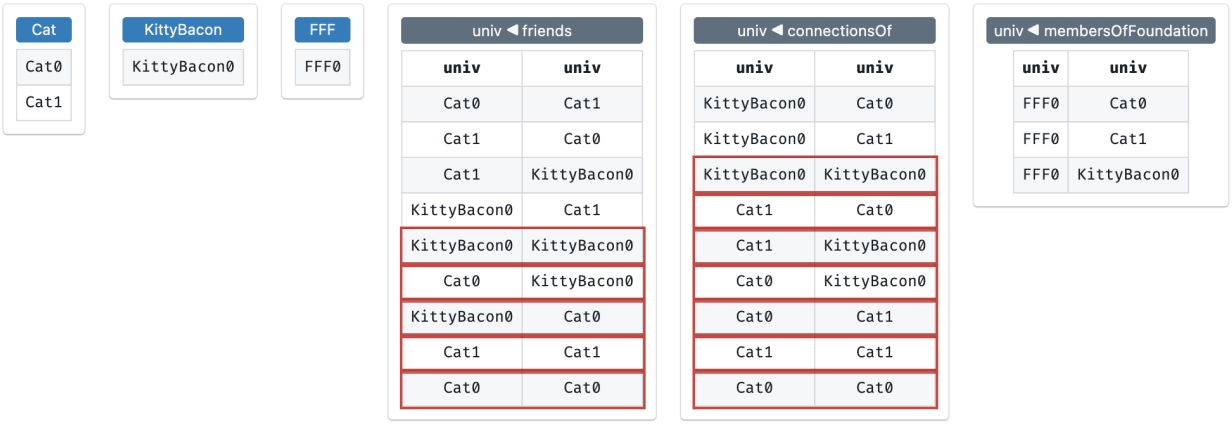
univ ◀ membersOfFoundation

univ	univ
FFF0	Cat0
FFF0	Cat1
FFF0	KittyBacon0

This is telling us the following information:

- `friends` relation
  - The edges `(Cat0, Cat1), (Cat1, Cat0), (Cat1, KittyBacon0),` and `(KittyBacon0, Cat1)` are true in the instance and can't be removed without violating the spec or making other changes to the instance.
- `connectionsOf` relation
  - The edges `(KittyBacon0, Cat0)` and `(KittyBacon0, Cat1)` are true in the instance and can't be removed without violating the spec or making other changes to the instance.
- `membersOfFoundation` relation
  - The edges `(FFF0, Cat0), (FFF0, Cat1),` and `(FFF0, KittyBacon0)` are true in the instance and can't be removed without violating the spec or making other changes to the instance.

Negative Local Necessity:



Cat

Cat
Cat0
Cat1

KittyBacon

KittyBacon
KittyBacon0

FFF

FFF
FFF0

univ ◀ friends

univ	univ
Cat0	Cat1
Cat1	Cat0
Cat1	KittyBacon0
KittyBacon0	Cat1
KittyBacon0	KittyBacon0
Cat0	KittyBacon0
KittyBacon0	Cat0
Cat1	Cat1
Cat0	Cat0

univ ◀ connectionsOf

univ	univ
KittyBacon0	Cat0
KittyBacon0	Cat1
KittyBacon0	KittyBacon0
Cat1	Cat0
Cat1	KittyBacon0
Cat0	KittyBacon0
Cat0	Cat1
Cat1	Cat1
Cat0	Cat0

univ ◀ membersOfFoundation

univ	univ
FFF0	Cat0
FFF0	Cat1
FFF0	KittyBacon0

This is telling us the following information:

- `friends` relation
  - The edges `(KittyBacon0, KittyBacon0), (Cat0, KittyBacon0), (KittyBacon0, Cat0), (Cat1, Cat1),` and `(Cat0, Cat0)` are false in the instance and can't be added without violating the spec or making other changes to the instance.

        - Note that negative local necessity is found by checking if a given tuple breaks the spec one at a time (therefore, even if there seems to be an edge from `(Cat0, KittyBacon0)` and `(KittyBacon0, Cat0)` in the friends relation, the tool is actually just testing adding the edge `(Cat0, KittyBacon0)` (which breaks the spec because the edge `(KittyBacon0, Cat0)` is not already present in the instance)).
- `connectionsOf` relation
    - The edges `(KittyBacon0, KittyBacon0)`, `(Cat1, Cat0)`, `(Cat1, KittyBacon0)`, `(Cat0, KittyBacon0)`, `(Cat0, Cat1)`, `(Cat1, Cat1)`, and `(Cat0, Cat0)` are false in the instance and can't be added without violating the spec or making other changes to the instance.

How to get Local Necessity On Your Own Instances?

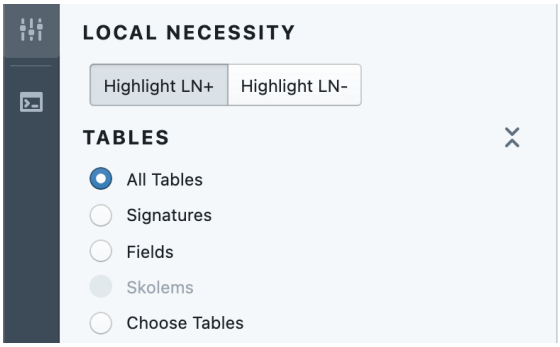
You can see the local necessity of an instance both in the "Graph" and "Table" view of the visualizer. However, for the context of this lab, **we recommend utilizing the Table View to see local necessity.**

Steps to get Local Necessity in Table View

(1) Run your code!

(2) In Sterling, switch from Graph View to Table View by pressing the "Table" button on the top toolbar.

(3) Click on the first button in the left-hand-side toolbar  . You will see the following window pop-up:



(4) On the **Local Necessity** section, press on the **Highlight LN+** or **Highlight LN-** buttons to see the LN+ or LN- of your instance.

(a) LN+ will highlight the table entries in **green** while LN- will highlight the table entries in **red**.

Setup

Start by opening a new file in DrRacket. At the very top of the file, make sure to include the following line, where <email> is your Gradescope email. If there is already a `#lang` line, you should replace it with this one.

```
#lang forge "lab1" "<email>"
```

For example, one of our TA's lab looks as follows:

```
#lang forge "lab1" "x0f00iexvy4yoidl@gmail.com"
```

Underneath the line that you just added, include the following line of code:

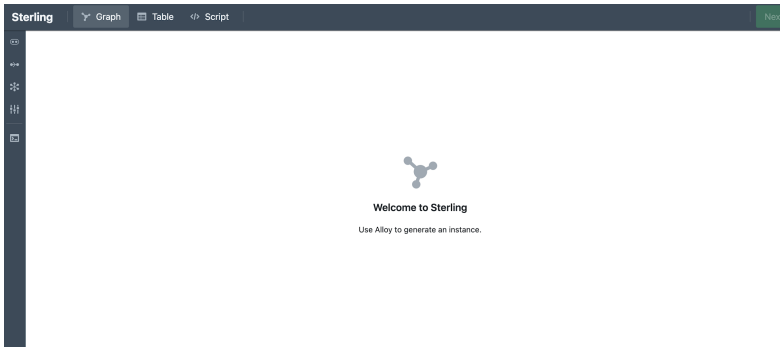
```
option local_necessity on
```

You might see a message at the bottom of your Forge screen saying "uncaught exn: "Email verification failed." or "call-with-output-file: forbidden (write delete) access to …". Feel free to ignore these.

Afterwards, run your code. You might be prompted to write your email address and the name of the assignment once again. If you are, write the same email as above (without quotations) and in the assignment name write lab1 (without quotations as well).

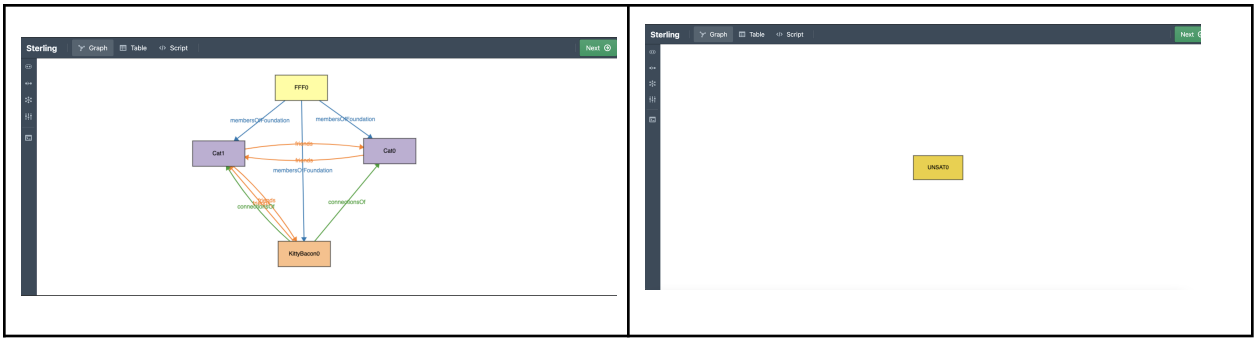
Note on Running Your Code

Whenever you run your code, a window like the one below will appear in your Internet browser. This program is called *Sterling*.



After waiting a few seconds (~15), a graph will show up in that window if the spec is satisfiable. In the case that the spec is unsatisfiable, a box saying `UNSAT0` will show up.

Satisfiable Spec	Unsatisfiable Spec



We Are All Set!

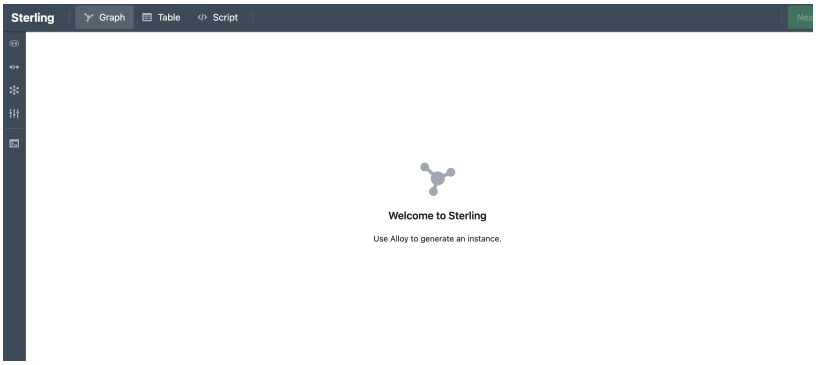
Begin with the following Forge signature describing `Cat`s, each of which has a set of `friends`:

```
sig Cat {
  friends : set Cat
}
run {}
```

As crucial as the friendship of kittens is, we're using it here as a playful framing of very real problems in computer science.

Averting Catastrophe

Run this Forge model. You will see that the visualizer will pop-up a window like the one below. In order for the instances to show up, **please wait a few seconds (~15)**. It is normal for Forge to take a while to show you instances. **In general, throughout the lab, if the terminal in Forge is printing some output, the instance will not show up in the visualizer until the printing is done.**



The window will update itself to show you an instance if the spec is valid or will show an `UNSAT0` box if the spec is unsatisfiable. **Note that you will not be able to see the local necessity for the instances shown in this initial run.**

View several instances in the visualizer by clicking *Next*. Again, the new instances will show up after a few seconds (~15). This model has a few unpleasant characteristics that we would like to constrain.

No Lonely Kitten

Make sure that all Cats have at least one friend. Using the predicate below as a template, replace `...` with your implementation:

```
pred NoLonelyKittens {
  …
}
```

Cats are Friends With *Other* Kittens

Some of the cats are claiming themselves as friends! Introduce a fact ensuring that the friends of a cat do not include itself. Using the predicate below as a template, replace `...` with your implementation:

```
pred OutsideFriends {
  …
}
```

Friendship Isn't A One-Way Street

Well, *Friendship* might be a one-way street in Providence, but these cats aren't from around here. Visualize an instance of your model so far, and note that the arrows connecting instances of `Cat` may be one-directional. Introduce a predicate `TwoWayStreet` ensuring that if one cat considers another cat as a friend, the other cat reciprocates the friendship:

```
pred TwoWayStreet {
  …
}
```

You can combine these all together into one predicate:

```
pred friendship {
      NoLonelyKittens
      OutsideFriends
      TwoWayStreet
}
```

Try running this predicate with `run friendship` to see all the cats! It is normal for the first instance to be empty, try clicking “Next” to see other instances!

Kitty Bacon

There's one special `Cat` in our universe, an actor by the name of Kitty Bacon! Add this signature to your model:

```
one sig KittyBacon extends Cat { connectionsOf : set Cat }
```

If you view an instance of this model, you should see exactly one `Cat` with the name `KittyBacon`.

Degrees of Kitty Bacon

Kitty Bacon is very connected. All kitties are in his connections. Add the following predicate:

```
pred ConnectedKittyBacon {
    --connectionsOfKittyBacon will be defined soon!
    connectionsOfKittyBacon
    Cat - KittyBacon = KittyBacon.connectionsOf
}
```

`ConnectedKittyBacon` depends on the relation `KittyBacon.connectionsOf`, which should consist of the union of `KittyBacon`'s `friends` of `friends` (not including `KittyBacon` themself) and `KittyBacon`'s `friends`.

It turns out that doing `KittyBacon.friends.friends` gets you `KittyBacon`'s `friends` of `friends`! Play around with this in the evaluator to see why that is. With that in mind, complete the definition below. **Make sure to write this predicate above** `ConnectedKittyBacon`:

```
pred connectionsOfKittyBacon {
    friendship
    KittyBacon.connectionsOf = ...
}
run {ConnectedKittyBacon and connectionsOfKittyBacon}
```

Look at the visualization of your instance. Are *all* cats connected to `KittyBacon` via their network of friends? If you got `UNSAT`, double-check that your definition of `KittyBacon.connectionsOf` isn't including too many cats.

A common pattern in Forge is to test a suspicion by phrasing it differently in a second predicate, and checking if both predicates are equivalent. An alternative way to express connected-via-friends is the *transitive closure operator*, `^`.

`KittyBacon.^friends` includes `friends`, `friends` of `friends`, `friends` of `friends` of `friends`, and so on. Include the following predicate in your spec:

```
pred SuperConnected {
    connectionsOfKittyBacon
    Cat - KittyBacon in KittyBacon.^friends
}
```

Two predicates are equivalent (up to bound only) if and only if (`iff`) one implies the other. Fill in the predicate below:

```
pred ConnectedKittyBacon_equals_SuperConnected {
  …
}
```

Finally, check the predicate:

```
check ConnectedKittyBacon_equals_SuperConnected for exactly 3 Cat
```

If you run this `check` and a graph pops-up in the visualizer, it means that you got a counter-example. If the visualizer shows an `UNSAT0` box, then your `check` passed. Were any counter-examples found?

What happens if you increase the scope of the model to include an additional `Cat`s?

```
check ConnectedKittyBacon_equals_SuperConnected for exactly 4 Cat
```

Modify KittyBacon.`connectionsOf` to include their `friends` of `friends` of `friends` and check for counterexamples again. If none are found, increase the scope of the `check` to `exactly 5 Cat`. Is it possible to modify KittyBacon.`connectionsOf` using only the `+` operator so that for any number of degrees of separation, and for any number of cats, no counterexample to this predicate can be found? If not, why? Write your response in a comment.

The Fancy Feline Foundation

Because `KittyBacon` is so fancy, he would like to create a foundation with all of his connections. Add the following signature to your model:

```
one sig FFF  { membersOfFoundation: set Cat }
```

All of `KittyBacon`'s connections are part of the `FancyFelineFoundation`:

```
pred FancyFelineFoundation {
```

```
    ConnectedKittyBacon
    FFF.membersOfFoundation = KittyBacon.connectionsOf
}
```

Since KittyBacon is such a fancy feline (he started the FancyFelineFoundation, after all), we can check whether KittyBacon is part of the FancyFelineFoundation:

```
pred KittyBaconIsAFancyFeline {
    FancyFelineFoundation implies KittyBacon in FFF.membersOfFoundation
}
```

Now, add the following check:

```
check KittyBaconIsAFancyFeline for exactly 3 Cat
```

What do you see when you run your check? Do you get any counterexamples? Is KittyBacon ever a part of the FancyFelineFoundation?

Tip: You can always look at local necessity and see whether it is necessary for KittyBacon to be (or not to be) in the FancyFelineFoundation. If you can't remember how to see the Local Necessity for an instance, go back to page 4 to 5.

If KittyBacon is never a part of the FancyFelineFoundation, make sure to modify your code to include him (after all, he started it!).

After your fix, run your `check` once again and see if there are any counterexamples. If there are none (meaning that no graphs show up), **please write what your fix was in a comment and your reasoning behind it**.

Now, we are going to add a final predicate to see a graph representing our entire model (including KittyBacon in the FancyFelineFoundation). Please add the following predicate:

```
pred seeAllCats {
    FancyFelineFoundation
    KittyBacon in FFF.membersOfFoundation
}
```

You might be wondering why we are adding this. Well, the fix that we added might have solved the problem of KittyBacon not being a part of the FancyFelineFoundation, but **did our fix lead to any unintended consequences?** This is very common in programming! While normally we would also write a few example tests, in the scope of this lab we are just going to run the predicate and see a couple of instances to make sure that we did not break any properties that should hold.

Include the following run statement in your spec:

```
run seeAllCats for exactly 3 Cat
```

Make sure to keep the visualizer window open from your run since you will need to show it to the TAs before check-off.

Check-Off

Call a TA over to talk about your responses.