Data Structure Repair Using Goal-Directed Reasoning

Brian Demsky
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

Model-based data structure repair is a promising technique for enabling programs to continue to execute successfully in the face of otherwise fatal data structure corruption errors. Previous research in this field relied on the developer to write a specification to explicitly translate model repairs into concrete data structure repairs, raising the possibility of 1) incorrect translations causing the supposedly repaired concrete data structures to be inconsistent, and 2) repaired models with no corresponding concrete data structure representation.

We present a new repair algorithm that uses goal-directed reasoning to automatically translate model repairs into concrete data structure repairs. This new repair algorithm eliminates the possibility of incorrect translations and repaired models with no corresponding representation as concrete data structures. Unlike our old algorithm, our new algorithm can also repair linked data structures such as a list or a tree.

1. INTRODUCTION

Programs usually make assumptions about the states of the data structures that they manipulate. A software error or some other event may cause the data structures to violate consistency assumptions that the software relies on. Data structure repair is a useful technique for restoring consistency properties, enabling the program to continue to execute successfully. Our previous work [9, 10, 8] introduced a model-based approach in which the developer uses a specification language to identify the required data structure consistency properties and provided an experimental evaluation of this approach in which the repair algorithm was used to improve the reliability of four different benchmarks.

This model-based approach involves two views: a concrete view of the data structures as they are represented in the memory and an abstract view that models the data structures as sets of objects and relations between objects. A set of model definition rules translates the concrete data structures to the sets and relations in the abstract model. The key consistency constraints are expressed using the sets and relations in this model. There are three challenges in making this approach effective: 1) maintaining a correspondence between the abstract model and the concrete data structures, 2) generating a set of repairs that is sufficient to repair any error, and 3) ensuring that all repairs terminate.

Our previous work [9, 10, 8] performed repairs on the abstract model and relied on a set of user-defined *external consistency constraints* to faithfully translate these model repairs to the actual data structures. While this approach automates much of the repair process, the presence of the external consistency constraints has several undesirable properties:

- An error in the external consistency constraints may cause the repair algorithm to fail to correctly translate the model repair into data structure updates. In this case the data structures would remain inconsistent even after the repair.
- The repair algorithm may generate abstract models that can not be represented as concrete data structures. To avoid this possibility, the developer may need to add additional model constraints that prevent the repair process from constructing such a model.

Our new algorithm replaces the external consistency constraints with a goal-directed reasoning algorithm on the model definition rules. The new approach has several advantages over the previous approach:

- It eliminates the possibility of errors in the external consistency constraints and guarantees that repairs are correctly translated from the model to the concrete data structures.
- It eliminates the possibility that the repair algorithm may produce a model with no corresponding concrete data structure representation.
- It provides enhanced support for linked data structures, enabling the new repair algorithm to add or remove objects from the data structures as required to satisfy the consistency constraints.

1.1 Repair Algorithm Generator

A set of model definition rules defines a translation from the concrete data structures to an abstract representation. Each rule consists of a quantifier, a guard, and an inclusion constraint that specifies an object (or a tuple) to include in a specific set (or relation). These rules place objects into sets based on criteria such as the values of the fields in the object and the reachability of the object from other objects. The key consistency constraints are expressed using the sets

*This research was supported in part by a fellowship from the Fannie and John Hertz Foundation, DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, and NSF Grant CCR00-63513.

and relations in the abstract model. Our specification language supports constraints between the values of variables and object fields, on the referencing relationships between objects, and on the absence or presence of certain objects.

When invoked, our repair algorithm constructs the model and examines it to find any inconsistencies. Whenever the repair algorithm discovers an inconsistency, it selects an appropriate model repair action to repair the inconsistency in the model. Our repair algorithm generator uses goal-directed reasoning to map model repair actions to concrete data structure updates. To implement a model repair action that removes an object from a given set, for example, the algorithm generator analyzes the model definition rules to find all rules whose inclusion constraint may cause the object to be inserted into the set. It then analyzes the guards and the quantifiers of the rules to extract a set of data structure properties whose satisfaction ensures that no rule specifies that the object should be a member of that set. Finally, it computes and applies (as necessary) a set of data structure updates that force all of these properties to hold. The effect is to remove the object from the set. Note that there may also be potentially undesirable side effects which cause additional inconsistencies. The algorithm must then apply additional repairs to correct these inconsistencies.

At each step in the repair process, the repair algorithm may be forced to choose between several alternatives — in general, there may be several distinct sets of model repair actions that cause a given violated constraint to become satisfied, several distinct sets of data structure updates that implement a given model repair action, and several different ways to eliminate any undesirable side effects of the data structure updates. A naive repair strategy can easily fail to terminate — it can get into a loop in which it repeatedly repairs a violated constraint, only to have the constraint repeatedly invalidated as a side effect of a subsequent action taken to repair another constraint violated as a side effect of the first repair action.

Our algorithm uses a *repair dependence graph* to reason about the termination of the repair process. The nodes in this graph represent constraints and repair actions. The edges represent dependences between the constraints, repair actions, and choices in the repair process. The absence of certain cycles in the graph ensures that all repairs will terminate. In addition to analyzing the graph to determine termination, our algorithm may also (when possible and subject to certain graph consistency conditions) remove nodes or edges to eliminate undesirable cycles. These removals constrain the actions of the repair algorithm and ensure that it will never choose a repair strategy that leads to an infinite repair loop.

1.2 Contributions

This paper makes the following contributions:

- **Basic Repair Approach:** It presents an approach that allows the developer to use an abstract model to express important data structure consistency properties. Violations of these properties are repaired by automatically translating model repairs back through the model definition rules to automatically derive a set of data structure updates that implement the repair.

- **Repair Translation:** It presents an algorithm that uses goal-directed reasoning to translate repairs in the abstract model back through the model definition rules to derive a set of data structure updates that implement the repair.

- **Repair Dependence Graph:** It introduces the repair dependence graph, which captures dependences between consistency constraints, repair actions, and choices in the repair process. This graph supports formal reasoning about the effect of repairs to both the model and the data structures.

  It also presents a set of conditions on the repair dependence graph. These conditions identify a class of cycles whose absence guarantees that all repairs will successfully terminate. It also presents an algorithm that, when possible, removes nodes and edges in the graph to eliminate problematic cycles. These removals prevent the repair algorithm from choosing repair strategies that may not terminate.

- **Linked Data Structures:** It presents support for adding and removing objects from linked data structures. This support enables the developer to express constraints on membership in a linked data structure, and enables the repair algorithm to regenerate damaged backpointers in linked data structures.

2. EXAMPLE

We next present an example that illustrates the operation of our repair algorithm. We start with the concrete data structure in our example. Figure 1 presents the structure definitions for the example data structure; these definitions give the physical layout of the objects comprising the concrete data structures. The `process` object participates in both a tree and a list structure. It contains a `left` pointer, which references its left child in the tree; a `right` pointer, which references its right child in the tree; a `next` pointer, which references the next `process` object in a list; and the `active` flag, which indicates whether the `process` is active. [1] Figure 2 graphically presents a concrete instance of an (inconsistent) process list and tree.

```
struct process {
  process *left;
  process *right;
  process *next;
  bool active;
  ...
}
process *tree;
process *list;
```

Figure 1: Structure Definitions

2.1 Set and Relation Definitions

Figure 3 contains the definitions for the sets and relations in the model. Our model consists of the set `ActiveProcesses`, which contains the `process` objects in the list, and the set `Processes`, which contains the `process` objects in the tree. The `Next`, `Left`, `Right`, and `Active` relations model the values of the `next`, `left`, `right`, and `active` fields in the

[1] Our algorithm also supports mode conventional implementations of the data structure.

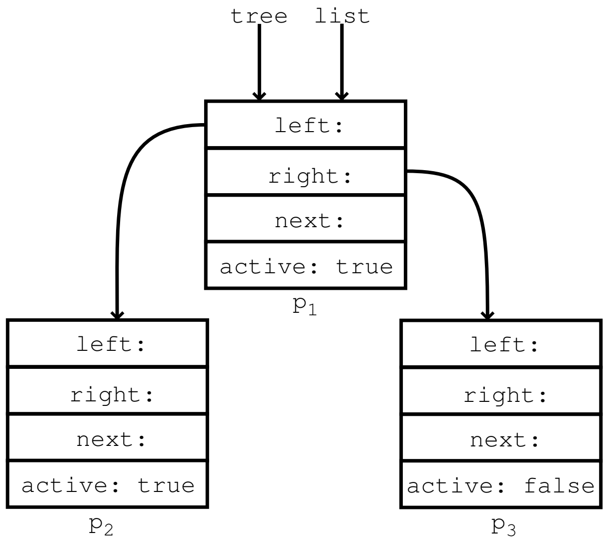

Figure 2: Inconsistent Process Structure

```
set ActiveProcesses of process
set Processes of process
relation Next: process -> process
relation Left: process -> process
relation Right: process -> process
relation Active: process -> bool
```

Figure 3: Set and Relation Definitions

`process` object. In this example, the relations closely parallel the contents of the data structure fields. In general, the relations tend to be more abstract and capture conceptual properties further removed from the concrete data structures.

2.2 Model Definition Rules

The model definition rules specify a translation from the concrete data structures to an abstract model. Conceptually, these rules specify how to traverse the data structures to build the sets and relations in the model. Furthermore, they provide the developer with a means to separate objects into different sets and to apply different model constraints to these different sets. Each rule specifies a quantifier that identifies the scope of the variables in the body. The body contains a guard and an inclusion condition. The inclusion condition specifies an object (or a pair) that must be in a specific set (or relation) if the guard is true. The least fixed point of the model definition rules applied to the concrete data structure generates the abstract model. The model definition rules for the example are given in Figure 4. The first rule specifies that the `process` object referenced by the `tree` pointer is in the `Processes` set. The second rule specifies that the `process` object referenced by the `list` pointer is in the `ActiveProcesses` set. The next two rules specify that the left and right children of any object in the `Processes` set are also in the `Processes` set. The next three rules construct the `Left`, `Right`, and `Active` relations to model the `left`, `right`, and `active` fields of objects in the `Processes` set. The eighth rule specifies that an object referenced by the `next` field of an object in the `ActiveProcesses` set is also in the `ActiveProcesses` set. The final rule constructs the `Next` relation to model the `next` field of objects in the `ActiveProcesses` set. The least fixed point of the model definition rules is evaluated on the concrete data structure shown in Figure 2 to construct the abstract model in Figure 5. The bold circles in the Venn diagram represent objects in the data structure, the edges represent the relations, and the large circles represent the sets.

```
1. tree != null => tree in Processes
2. list != null => list in ActiveProcesses
3. for p in Processes, p.left != null =>
     p.left in Processes
4. for p in Processes, p.right != null =>
     p.right in Processes
5. for p in Processes, p.left != null =>
     <p,p.left> in Left
6. for p in Processes, p.right != null =>
     <p,p.right> in Right
7. for p in Processes, true => <p,p.active> in Active
8. for p in ActiveProcesses, p.next !=null =>
     p.next in ActiveProcesses
9.  for p in ActiveProcesses, p.next !=null =>
     <p,p.next> in Next
```

Figure 4: Model Definition Rules

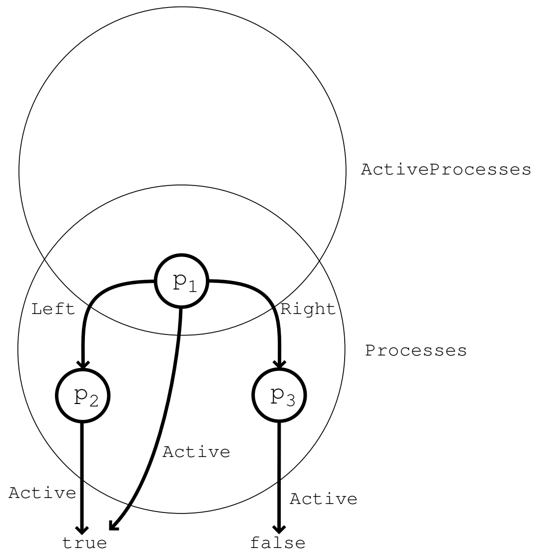

Figure 5: Inconsistent Abstract Model

2.3 Model Constraints

The model constraints identify the consistency properties that must hold in the model. Many of these rules use the `size` predicate, which is used both to relate the number of objects in a set to a value in a relation and to constrain the number of objects in a set. For example, the first constraint in Figure 6 ensures that each `process` object in the `ActiveProcesses` set has an in-degree of at most one. [2] The second constraint ensures that each `process` in the `Processes` set has an in-degree of at most one. The final constraint ensures that each `process` in the `Processes` set

[2]Note that we use the notation `Next.p` to indicate the image of `p` under the inverse of the `Next` relation; i.e., the set of all `m` such that $\langle$`m`, `p`$\rangle$ in `Next`.

either has its `active` flag set to false, or is in the `ActiveProcesses` set. Note that the `process` object labeled $\mathtt{p}_2$ in Figure 5 is in the `Processes` set, has the `active` flag set, and is not in the `ActiveProcesses` set. This violates the final consistency constraint, `for p in Processes, !p.Active or p in ActiveProcesses`.

```
for p in ActiveProcesses, size(Next.p)<=1
for p in Processes,
  (size(Left.p)<=1 and size(Right.p)=0) or
  (size(Left.p)=0 and size(Right.p)<=1)
for p in Processes, !p.Active or p in ActiveProcesses
```

Figure 6: Model Constraints

2.4 Previous Repair Algorithm

Our previous repair algorithm [9, 10, 8] constructs an abstract representation of the data structures to be repaired, performs repairs on this abstract representation, and then uses a set of *external consistency constraints* to translate the abstract repairs to the concrete data structure.

Figure 7 presents a set of external consistency constraints for the process example. [3] The constraints shown ensure that the `Left`, `Right`, `Next`, and `Active` relations are written to the `left`, `right`, `next`, and `active` fields of the `process` objects.

```
for <p1,p2> in Left, true => p1.left=p2
for <p1,p2> in Right, true => p1.right=p2
for <p1,p2> in Next, true => p1.next=p2
for <p,a> in Active, true => p.active=a
```

Figure 7: External Consistency Constraints for Previous Repair Algorithm

The model construction phase of our previous repair algorithm generates an identical abstract model to the model that appears in Figure 5. Our previous repair algorithm would detect that since object $\mathtt{p}_2$ is in the `Processes` set, has the `active` flag set, and is not in the `ActiveProcesses` set, it violates the consistency constraint `for p in Processes, !p.Active or p in ActiveProcesses`. As a result, our previous repair algorithm would add the object $\mathtt{p}_2$ to the `ActiveProcesses` set, generating the abstract model shown in Figure 8.

Since this model is consistent with all of the consistency constraints, the repair algorithm would then perform the updates specified by the external consistency constraints. Unfortunately, this fails to generate a consistent concrete data structure. The problem is that although the instance of the abstract model shown in Figure 8 satisfies the model constraints, it does not correspond to any concrete data structure. As a result, this instance of the abstract model cannot be faithfully translated to a concrete data structure by *any* set of external consistency constraints.

As this example shows, the correctness of the previous repair algorithm relies on the developer ensuring that the repair process generates a consistent model that corresponds to a concrete data structure (i.e., that the abstract model corresponds to the least fixed point of the model definition

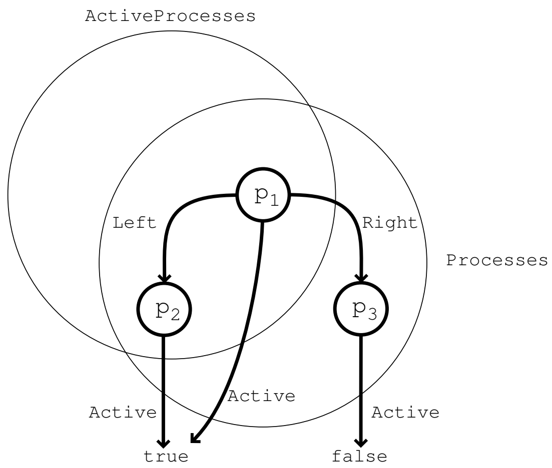

Figure 8: Incorrectly Repaired Abstract Model

rules evaluated on some concrete data structure) and that the external consistency constraints faithfully translate the consistent abstract model to a consistent data structure.

Our new repair algorithm eliminates all of these problems. It guarantees the correspondence of the abstract model to the least fixed point of the application of the model definition rules to the concrete data structures. This enables the repair algorithm to handle cases such as this example. Furthermore, the new algorithm requires less work on the part of the developer and eliminates the possibility of errors in the external consistency constraints. The result is an algorithm that can handle linked data structures, is easier to use, and eliminates the possibility of bad repairs due to errors in the external consistency constraints.

2.5 Repair Algorithm

Our new repair algorithm constructs an abstract representation of the data structures to repair, identifies model repairs to perform on the abstract representation, and then translates the model repairs into data structure updates. It then performs the data structure update on the concrete data structure, and recomputes the model to ensure that the model corresponds to the least fixed point of the model definition rules applied to the concrete data structures.

Notice that in Figure 5, the object $\mathtt{p}_2$ is in the `Processes` set, has its `active` flag set, and is not in the `ActiveProcesses` set. This violates the final constraint shown in Figure 6. To repair this violation, the algorithm must either remove $\mathtt{p}_2$ from the `Processes` set, set $\mathtt{p}_2$'s `active` flag to `false`, or add $\mathtt{p}_2$ to the `ActiveProcesses` set. [4] Assume that the algorithm chooses to add the object $\mathtt{p}_2$ to the `ActiveProcesses` set. An examination of the model definition rules reveals that the model definition rule, `for p in ActiveProcesses, p.next !=null => p.next in ActiveProcesses`, inserts objects into the `ActiveProcesses` set. Examination of this rule reveals that its guard is `p.next != null` and its inclusion condition is `p.next in ActiveProcesses`. Goal-directed reasoning makes it clear that the repair algorithm must set the

[3] Note that the external constraints are not used by the repair algorithm presented in this paper.

[4] Note that the repairs correspond to satisfying one of the conjunctions in the disjunctive normal form (DNF) of the constraint or to removing an object (or tuple) from the set (or relation) that the model constraint is quantified over.

next field of p_1 to p_2 and the next field of p_2 to null (since the previous value of p_2.Next was {}). After the next fields are updated, the algorithm recomputes the abstract model using a fixed point calculation.

Note that this data structure update has additional effects. Specifically, the tuple $\langle p_1, p_2 \rangle$ is added to the Next relation. At this point, the model is consistent and the repair process terminates. Figure 9 shows the abstract model after the repair algorithm has finished, and Figure 10 shows the concrete data structures after the repair algorithm has finished. Note that the model in Figure 9 contains a reference in the Next relation from the object p_1 to the p_2 that is absent in Figure 8.

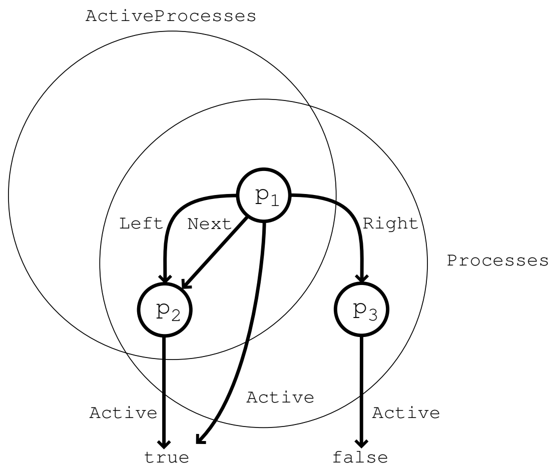

Figure 9: Correctly Repaired Abstract Model

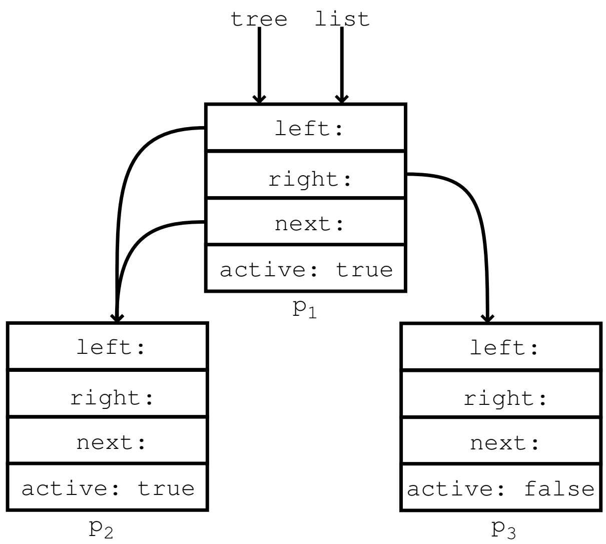

Figure 10: Consistent Process Structure

2.6 Repair Algorithm Generation

As illustrated in the preceding section, a successful repair algorithm must perform several steps: 1) traverse the model to find an inconsistency, 2) analyze the model definition rules to find updates to the concrete data structures that will eliminate the inconsistency, 3) perform the updates, 4) repeat to eliminate any remaining or newly introduced inconsistencies, and 5) terminate. The key challenges are finding the data structure updates and reasoning about termination. As described above, the algorithm uses goal-directed reasoning to extract conditions from the model definition rules; the execution of data structure updates that satisfy these conditions implements the model repair.

The algorithm uses the repair dependence graph to reason about termination. The nodes in this graph model constraints and repair actions; the edges represent dependences. In our example the graph contains (in addition to other edges as described below in Section 7) nodes for each model definition rule, nodes for repair actions, and nodes for the conjunctions in the disjunctive normal form (DNF) of the model constraints such as `for p in ActiveProcesses, size(Next.p)<= 1`. The conjunction nodes correspond to the different options for satisfying a given model constraint. The graph contains an edge from the node representing the conjunction `size(Next.p)<=1` to the node representing the action of removing a tuple from the Next relation. This edge captures the fact that the repair algorithm may choose to repair a violation of this constraint by removing a tuple from the Next relation. In general, the graph edges model the propagation of the effects of repair actions. The absence of certain kinds of cycles ensures that the effect of every repair action will propagate only a finite distance, ensuring that repairs terminate.

3. SPECIFICATION LANGUAGE

Our specification language consists of the structure definition language, model definition language, and model constraint language. The structure definition language is similar to that of C, but supports a wider range of primitive data types.

3.1 Model Definition Language

The model definition language allows the developer to declare the sets and relations in the model and to specify the rules that define the model. The model definition rules define a translation from the concrete data structures into the abstract model. Figure 11 presents the grammar for the set and relation declarations and the model definition rules. The model constructed by the repair algorithm is the least fixed point generated by the application of this set of model definition rules.

3.2 Model Constraint Language

Figure 12 presents the grammar for the model constraint language. Each constraint consists of an optional quantifier Q followed by a body B. The body uses logical connectives (and, or, not) to combine basic propositions P. We treat undefined values in this semantics by appropriately extending arithmetic operations to work with undefined values and logical operations to work with maybe according to the laws of three-valued logic.

In many cases, relations are used as functions. To ensure that these uses are well formed, the repair algorithm requires the specification to include additional constraints that constrain the relations to be functions. For example, our system requires that specifications containing expressions of the form $E.R$ also contain another constraint either

$$
\begin{array}{rcl}
D & := & \texttt{set } S \texttt{ of } T \mid S \texttt{ partition } (S,)^* \mid S \texttt{ subset } (S,)^* \mid \\
 & & \texttt{relation } R\texttt{:}S\texttt{->}S \mid \texttt{relation } R\texttt{:}T\texttt{->}T \\
M & := & (Q,)^*\ G\texttt{=>}I \\
Q & := & \texttt{for } V \texttt{ in } S \mid \texttt{for } \langle V,V\rangle \texttt{ in } R \\
G & := & G \texttt{ and } G \mid G \texttt{ or } G \mid \texttt{!}G \mid (G) \mid P \\
P & := & FE\texttt{=}E \mid FE\texttt{<}E \mid FE\texttt{<=}E \mid FE\texttt{>=}E \mid FE\texttt{>}E \mid \\
 & & \texttt{true} \mid E \texttt{ in } S \mid \langle E,E\rangle \texttt{ in } R \\
I & := & FE \texttt{ in } S \mid \langle FE,FE\rangle \texttt{ in } R \\
E & := & FE \mid \mathit{number} \mid \mathit{string} \mid E\texttt{+}E \mid E\texttt{-}E \mid E\texttt{*}E \mid E\texttt{/}E \\
FE & := & V \mid V\texttt{.field}
\end{array}
$$

Figure 11: Model Definition Language

$$
\begin{array}{rcl}
C & := & Q,C \mid B \\
Q & := & \texttt{for } V \texttt{ in } S \mid \texttt{for } \langle V,V\rangle \texttt{ in } R \\
B & := & B \texttt{ and } B \mid B \texttt{ or } B \mid \texttt{!}B \mid (B) \mid P \\
P & := & VE\texttt{=}E \mid VE\texttt{<}E \mid VE\texttt{<=}E \mid VE\texttt{>}E \mid \\
 & & VE\texttt{>=}E \mid V \texttt{ in } SE \mid \texttt{size}(SE)\texttt{=}c \mid \\
 & & \texttt{size}(SE)\texttt{>=}c \mid \texttt{size}(SE)\texttt{<=}c \\
VE & := & V.R \mid VE.R \\
E & := & V \mid \mathit{number} \mid \mathit{string} \mid E\texttt{+}E \mid E\texttt{-}E \mid E\texttt{*}E \mid E\texttt{/}E \mid \\
 & & E.R \mid \texttt{size}(SE) \mid (E) \\
SE & := & S \mid VE \mid R.V
\end{array}
$$

Figure 12: Model Constraint Language

of the form $\texttt{size}(V.R) = 1$ or $V.R = E'$ where V quantifies over a set containing all the possible sets of values for E.

3.3 Desugaring Partition and Subset Constraints

The repair algorithm enforces the partition constraints and subset constraints by desugaring these constraints into model constraints and model definition rules. The repair algorithm first analyzes the partition and subset constraints to find cyclic dependencies. If the repair algorithm discovers a cyclic dependency, the repair algorithm simply collapses all sets in the cyclic dependency to a single set. Then the repair algorithm removes any constraints that are discovered to be extraneous as a result of collapsing sets in a cyclic dependency to a single set.

As a result, we can safely assume that the partition and subset constraints do not include any cycles. We then desugar the partition constraint $\texttt{S partition S}_1\texttt{, ..., S}_n$ into the model constraint: $\texttt{for } V \texttt{ in } S, (V \texttt{ in } S_1 \texttt{ and !}V \texttt{ in } S_2 \texttt{... and !}V \texttt{ in } S_n) \texttt{ or } (\texttt{!}V \texttt{ in } S_1 \texttt{ and } V \texttt{ in } S_2 \texttt{... and !}V \texttt{ in } S_n) \texttt{ or ... or } (\texttt{!}V \texttt{ in } S_1 \texttt{ and !}V \texttt{ in } S_2 \texttt{... and } V \texttt{ in } S_n)$.

Furthermore, if the subset or partition constraints imply $\forall s, s \in S_1 => s \in S_2$, then a new model definition rule will be created for any model definition rule that adds an element to S_1; this new rule will be the original rule modified to add the same element to S_2.

We optionally allow relation declarations to declare domain and range sets. We desugar relation declarations of the form $\texttt{relation } R : S_1\texttt{->}S_2$ into model constraints of the form: $\texttt{for } \langle V_1, V_2\rangle \texttt{ in } R, (V_1 \texttt{ in } S_1) \texttt{ and } (V_2 \texttt{ in } S_2)$. In some cases, this model constraint is not necessary. For example, our algorithm does not generate the model constraint for S_i if for each model definition rule of the form $(Q,)^*\ G \Rightarrow \langle FE_1, FE_2\rangle \texttt{ in } R$ there is a model definition rule of the form $(Q,)^*\ G \Rightarrow FE_1 \texttt{ in } S_i$.

4. MODEL CONSTRUCTION

The model definition rules define a translation from the concrete data structures to the abstract model. The model construction phase constructs the abstract model by computing the least fixed point of the model definition rules applied to the concrete data structure. Finally, the model construction algorithm keeps track of the memory layout to ensure that the concrete data structures are physically well formed (that they reside in allocated memory and that they don't illegally overlap).

4.1 Denotational Semantics

Figure 21 in Appendix C gives the denotational semantics $\mathcal{R}[M]\ h\ l\ m$ of a single rule C. A model m is a mapping from set names and relation names to the corresponding sets of objects or relations between objects. This mapping is represented using a set of tuples. We define $m(s)$ to be the set $\{\langle v, s\rangle \mid \langle v, s\rangle \in m\}$. The set h models the heap in the running program using a set of tuples representing the references in the heap. The set h contains tuples that represent a mapping of each legal pairing of object and field; or object, field, and integer index to exactly one $HeapValue$. Given a set of concrete data structures h, a naming environment l that maps variables to data structures or values, and a current model m, $\mathcal{R}[M]\ h\ l\ m$ is the new model after applying the rule to m in the context of h and l. Note that l provides the values of both the program variables that the rules use to reference the concrete data structures and the variables bound in the quantifiers.

Each model definition contains a set of model definition rules $M_1, ..., M_n$. Given a model containing these rules, a set of concrete data structures h, and a naming environment l for the program variables , the model is the least fixed point of the functional $\lambda m.(\mathcal{R}[M_1]\ h\ l) \ldots (\mathcal{R}[M_n]\ h\ l\ m)$.

4.2 Negation and the Rule Dependence Graph

The presence of negation in the model definition language complicates the computation of the least fixed point. For example, negation makes it possible for a rule to specify that an object is in a given set only if another object is not in another set. We address this complication by requiring the set of model definition rules to have no cycles that go through rules with negated inclusion constraints in their guards.

We formalize this constraint using the concept of a *rule dependence graph*. There is one node in this graph for each rule in the set of model definition rules. There is a directed edge between two rules if the inclusion constraint from the first rule has a set or relation used in the quantifiers or guard of the second rule. If the graph contains a cycle involving a rule with a negated inclusion constraint, the set of model definition rules is not well founded and we reject it. Given a well-founded set of constraints, our model construction algorithm performs one fixed point computation for each strongly connected component in the rule dependence graph, with the computations executed in an order compatible with the dependences between the corresponding groups of rules.

4.3 Pointers

Depending on the declared type in the corresponding structure declaration, an expression of the form E.`f` in a model definition rule may be a primitive value (in which case E.`f` denotes the value), a nested `struct` contained within E (in which case E.`f` denotes a reference to the nested `struct`), or a pointer (in which case E.`f` denotes a reference to the `struct` to which the pointer refers). It is of course possible for the data structures to contain invalid pointers. We next describe how we extend the model construction algorithm to deal with invalid pointers.

First, we instrument the memory management system to produce a trace of operations that allocate and deallocate memory (examples include `malloc`, `free`, `mmap`, and `munmap`). We augment this trace with information about the call stack and segments containing statically allocated data, then construct a map that identifies valid and invalid regions of the address space.

We next extend the model construction algorithm to check that each `struct` accessed via a pointer is valid before it inserts the `struct` into a set or a relation. All valid `struct`s reside completely in allocated memory. In addition, if two `struct`s overlap, one must be completely contained within the other and the declarations of both `structs` must agree on the format of the overlapping memory. This approach ensures that only valid `struct`s appear in the model. If two data structures illegally overlap, the repair algorithm nullifies the reference to one of the data structures. This guarantees that write operations to one data structure will not corrupt the other data structure, and that the model construction algorithm will generate the same fixed-point in the future.

Our model construction algorithm is coded with explicit pointer checks so that it can traverse arbitrarily corrupted data structures without generating any illegal accesses. It also uses a standard fixed point approach to avoid becoming involved in an infinite data structure traversal loop.

5. INCONSISTENCY REPAIR

The inconsistency detection algorithm iterates over all values of the quantified variables in the model constraints, evaluating the body of the constraint for each possible combination of the values. [5] If the body evaluates to false, the algorithm has detected a violation and has computed a set of bindings for the quantified variables that make the constraint false. We assume the violated constraint is in disjunctive normal form (disjunctions of conjunctions of basic propositions). Conceptually, a successful repair performs the following steps:

- **Conjunction Selection:** Satisfying all of the basic propositions in any of the constraint's conjunctions will ensure that the constraint is satisfied. The first step is therefore to select a conjunction to satisfy.
- **Model Repair:** Each basic proposition has a set of model repair actions that, when performed, ensure that the basic proposition is satisfied. The next step is therefore to perform these repair actions.
- **Data Structure Updates:** Each model repair action is implemented by a set of data structure updates; the specific set of updates is determined by the model definition rules. The next step analyzes these rules to perform the translation, then executes the updates.
- **Compensation:** The updates may have side effects that cause the model to change in undesirable ways, specifically by adding objects to sets or tuples to relations. It is sometimes possible to prevent these changes by performing additional compensation updates that falsify the guards in the model definition rules that caused the additions to take place.
- **Model Update:** The next step is to update the model to reflect the effect of the concrete data structure updates.

At this point the algorithm has repaired the violated constraint. However, the updates may have caused other constraints to become violated. The algorithm therefore reevaluates the constraints and repairs any remaining violations. The key issue is whether this process terminates.

We formulate the termination analysis as a graph problem. We build a graph that captures the dependences between the model constraints, the model definition rules, the repair actions, and the choices available to the algorithm. The absence of certain kinds of cycles in this graph guarantees that all repairs will terminate. When possible, our algorithm prunes choices to eliminate problematic cycles in the graph.

5.1 Model Repair Actions

We next discuss the model repair actions that repair violated basic propositions. The action depends on the form of the proposition. For size propositions such as $\texttt{size}(SE) = c$ the repair algorithm simply adds or removes objects (or tuples) from the appropriate set (or relation) to satisfy the constraint. For inequality propositions such as $VE = E$ the repair algorithm calculates the value of E, then updates VE to satisfy the proposition. For inclusion propositions such as V `in` SE the repair algorithm simply adds or removes the specified object (or tuple) to or from the specified set (or relation).

The actions that add objects to sets must satisfy the partition and subset requirements of the model definition. A single object addition or removal may therefore trigger a cascading sequence of object additions or removals as the algorithm readjusts the model to satisfy these requirements.

5.2 Data Structure Updates

We next discuss how the algorithm translates model repairs into actions that correctly update the concrete data structures. Given a model repair that adds an object to a set (or a tuple to a relation), the algorithm finds all model definition rules with an inclusion constraint that may cause the object (or tuple) to be added to the set (or relation). The goal is to synthesize a set of data structure updates that cause the guard of one of these rules to be satisfied, which in turn ensures that the object (or tuple) is in the set (or relation).

We assume the guards are in disjunctive normal form. The algorithm chooses a rule, chooses one of the guards' conjunctions, then updates the data structures to ensure that all of the propositions in the conjunction are true. The specific data structure update depends on the form of the proposition. For inequality propositions such as $FE < E$, the

[5] The algorithm uses the denotational semantics given in Figure 22 in Appendix C to evaluate the model constraints.

algorithm computes E to generate a value that satisfies the proposition, then assigns this value to FE. For propositions of the form E `in` S or $\langle E, E \rangle$ `in` R, the algorithm calls itself (recursively) to generate the appropriate data structure updates. The termination analysis in Section 9 ensures that this recursion terminates.

The algorithm uses a similar strategy to implement repairs that remove an object (or tuple) from a set (or relation). But instead of choosing one conjunction from one model definition rule and satisfying all concrete propositions in that conjunction, it instead chooses a set of concrete propositions that include at least one concrete proposition from each conjunction of each model definition rule that could cause the object or tuple to appear in the set or relation. It then performs actions that falsify (as necessary) the conjunctions in this set.

5.2.1 Consistent Concrete Propositions

While processing the specification, the repair algorithm generates a set of concrete propositions that it satisfies to implement the repair. The repair algorithm must statically verify that these propositions will not be contradictory. If the set of concrete propositions associated with a given concrete data structure update contains more than one concrete proposition with the same field or variable on the left hand side, the static analysis rejects the concrete repair (and does not include it in the dependence graph) unless one of these three conditions is true:

- The two concrete propositions are the same.
- One concrete proposition is an equality proposition between FE and a value known not to be NULL (a quantification variable V) and the second concrete proposition is of the form $!FE = NULL$.
- One concrete proposition is of the form $FE = NULL$ and the second concrete proposition is an inequality proposition between FE and a value known not to be NULL (a quantification variable V).

Finally, the algorithm checks that there is no dependence cycle between propositions that use and define the same `struct` field or variable. The repair action will satisfy the propositions in order of their dependences.

5.2.2 Atomic Modifications

In some cases the algorithm may need to change the value of a tuple in a relation rather than removing the tuple then replacing it with a new tuple. Consider, for example, a model definition rule of the form `true => <v, v.f> in R`. There is no action that will remove `<v,v.f>` from `R`. In this case the algorithm changes `v.f` if the presence of the tuple with the old value of `v.f` causes the model to be inconsistent.

5.2.3 New Objects

The repair action may need a source of new objects to add to sets to bring them up to the specified size or to serve as wrapper objects. Any supersets of the set (as specified using the model definition language from Section 3.1) are one potential source. For primitive types, such as integers, the action can simply synthesize new values. For `structs`, memory allocation primitives are a potential source of new objects. [6] We allow the developer to specify which source to use and, in the absence of such guidance, use heuristics to choose a default source.

5.2.4 Recursive Data Structures

The repair algorithm discovers recursive data structures by searching for pairs of model definition rules: a base case model definition rule of the form $G \Rightarrow FE$ `in` S and a recursive case model definition rule of the form `for` V `in` $S, G' \Rightarrow FE'$ `in` S. The repair algorithm must consider two cases for adding an object to a recursive data structure: it can either add the object to the beginning of the recursive data structure or it can add the object after another object in the recursive data structure. If the repair algorithm adds the object O to the beginning of the data structure, the repair algorithm must satisfy $FE = O$, G, $G'[V/O]$ [7] , and $FE'[V/O] = O'$ where O' is the initial value of FE. If G is initially false, the last two propositions are replaced with $\neg G'[V/O]$. If the repair algorithm adds the object O after the object O', then the repair algorithm must satisfy $FE'[V/O'] = O$, $G'[V/O']$, $FE'[V/O] = O''$, and $G'[V/O]$ where O'' is the initial value of $FE'[V/O']$. If $G'[V/O']$ is initially false, the last two propositions are replaced with the proposition $\neg G'[V/O]$.

The repair algorithm must consider two cases for removing an object O from a recursive data structure. If object O is added by a rule of the form $G \Rightarrow FE$ `in` S, then the repair algorithm must satisfy $FE = FE'[V/O]$ and G unless $G'[V/O]$ was initially false, in that case it must satisfy $\neg G$. If object O is added by a rule of the form `for` V `in` $S, G' \Rightarrow FE'$ `in` S where $V = O'$, the repair algorithm must satisfy $FE'[V/O'] = O''$ and $G'[V/O']$ where O'' is the initial value of $FE'[V/O]$ unless $G'[V/O]$ was initially false, in that case it must satisfy $\neg G'[V/O']$.

This algorithm easily generalizes to handle specifications in which $G \Rightarrow FE$ `in` S is replaced with V' `in` $S', G \Rightarrow FE$ `in` S. This change does not effect the algorithm for adding or removing an object from the middle of a recursive data structure. To add an object to beginning of the recursive data structure, the algorithm finds an object in S' (or adds one if necessary) and binds V' to this object. To remove an object from the beginning of the recursive data structure, the algorithm finds the object binding for V that adds the header of the recursive data structure to S.

5.2.5 Improving Translation Precision

In some cases it is possible to further analyze the model definition rules to improve the precision of the translation of model repairs into data structure updates. Consider, for example, a set of concrete data structure updates whose intended effect is to add an object to a set in the abstract model. As described above, these updates satisfy the guard of the model definition rule that adds the object to the set. But these updates may also have unintended side effects. For example, they may affect the guards of other model definition rules, which may in turn cause other undesirable changes to the model.

[6] Note that the use of memory allocation means that the repair can fail if it runs out of memory. However, the termination analysis ensures that the repair process is not an infinite memory consumer.

[7] We use the notation $G'[V/O]$ to mean G' evaluated with V bound to O.

We therefore augment our translation algorithm to analyze the model definition rules to, when possible, perform additional compensation updates to eliminate the undesirable side effects. Given a model definition rule whose guard may be affected by the data structure update, our algorithm examines the rule's guard to derive additional updates that restore the original truth value of the guard. If, for example, the precondition of the guard is of the form G_1 `and` G_2, and the original update makes G_1 true, the compensation update will make G_2 false.

It is, of course, possible for compensation updates to have additional unintended side effects, which the algorithm eliminates with additional compensation updates. The termination analysis in Section 9 ensures that the algorithm never attempts to produce an infinite sequence of compensation updates. The net effect is to improve the precision of the translation by synthesizing larger, more precise data structure updates for each model repair.

6. DEVELOPER CONTROL OF REPAIRS

The repair algorithm often has multiple options for how to satisfy a given constraint; these options may translate into different repaired data structures. We recognize that some repair actions may produce more desirable data structures than other repair actions, and that the developer may wish to influence the repair process. We have therefore provided the developer with several mechanisms that he or she can use to control how the repair algorithm chooses to repair an inconsistent data structure.

6.1 Repair and Update Analysis

The termination analysis described in Section 9 generates the set of all possible model repairs and data structure updates. Our system enables the developer to specify conditions that model repairs and data structure updates must satisfy. For example, the developer can specify that certain fields or relations should not be modified, or that objects should not be removed from certain sets. The termination analysis would inform the developer whether a repair algorithm can be generated using only model repairs and updates that satisfy the developer imposed conditions. Furthermore, the developer can even review the automatically generated model repairs and updates to determine whether they are acceptable. This mechanism allows the repair process to be completely predictable and under the developer's control, yet retains the repair and termination guarantees provided by the automatically generated repair algorithms.

6.2 Repair Costs

The first mechanism is based on a repair cost associated with each basic proposition. At each step, the repair algorithm must choose one of several violated constraints to repair. Each constraint has a set of conjunctions; repairing any of these conjunctions will ensure that the constraint is satisfied. The repair of each conjunction, in turn, requires the execution of a repair action for each of its violated basic propositions. The repair algorithm sums the costs for each of the repair actions, then chooses the constraint and conjunction with the least repair cost.

The developer may specify the repair cost for each basic proposition. Developers may use this mechanism to, for example, bias the repair process toward preserving as much of the information present in the original inconsistent data structure as possible. One way to accomplish this goal is to assign higher costs to actions that remove objects from sets and pairs from relations and lower costs to actions that insert objects and pairs. The developer may also choose to assign lower costs to repair actions that change object fields or set flags and higher costs to repair actions that change the referencing relationships.

The choices made by our algorithm are easily isolatable from the repair code. It is possible for the developer to provide a procedure which would indicate which repair actions that the developer found acceptable. This procedure could select which conjunction to satisfy to repair a model constraint, which abstract repair to perform to satisfy a basic proposition, and which data structure update to use to perform an abstract repair. This mechanism gives the developer control over the choices made by the repair algorithm while maintaining the repair guarantees provided by the automatic repair algorithm.

6.3 Set Membership Changes

Some repair actions involve adding an object to a set. To execute such an action, the system must obtain a source for the object. The two standard sources are a memory allocator and another set of objects. The default choice is to use a memory allocator for structures and another set of objects for basic types such as integers and booleans. For each set in the model, we allow the developer to specify the source of objects for that set. We also allow the developer to similarly control the source of pairs added to relations.

6.4 Hand-Coded Repair Routines

In some cases, the developer may wish to provide a hand-coded repair algorithm. It is straightforward to extend our algorithm so that, for each constraint, the developer can specify a hand-coded repair procedure to invoke when the constraint is violated. When the hand-coded repair terminates, the system would verify that the constraint is satisfied, then (once again under developer control) optionally invoke its own standard repair algorithm if the hand-coded repair failed to satisfy the constraint.

7. THE REPAIR DEPENDENCE GRAPH

The repair algorithm constructs a *repair dependence graph* $\langle N, E\rangle$ to reason about the termination of the repair algorithm on a system of constraints. The nodes represent model conjunctions, repair actions, and model definition rules. The edges capture the dependences between the model constraints, repair actions, model definition rules, and choices in the repair process.

7.1 Nodes in Graph

The graph contains the following nodes:

- **Model conjunction nodes:** In disjunctive normal form, each model constraint C_i is of the form $C_i = Q_{i1}, ..., Q_{im} \bigvee_j^{j_{max}} C_{ij}$. There is one node N_{ij} for each conjunction C_{ij} in the model constraint C_i and an additional node $N_{ij'}$, where $j' = j_{max} + l$, for each quantifier Q_{il} in the model constraint.

- **Model repair nodes:** For each basic proposition C_{ijk} in each conjunction C_{ij} there is a set of nodes $\bigcup_l \{A_{ijkl}\}$ corresponding to the model repair actions that the repair algorithm may use to repair that basic proposition. There are also two model repair nodes A_r for each set and relation. One models insertions, the other removals. Edges from data structure update nodes and compensation update nodes (see below) to these nodes capture dependences in which the data structure update or compensation update action requires the insertion or removal of an object from a set or a tuple from a relation.

- **Data structure update nodes:** There is a set of data structure update nodes $\bigcup_m \{R_{ijklm}\}$ for each model repair node A_{ijkl} in the graph. These update nodes represent the concrete data structure updates that implement the repair. There is also a similar set of nodes $\bigcup_s \{R_{rs}\}$ for each model repair node A_r.
- **Increase and decrease scope nodes:** For each model definition rule $M_w = Q_w, (\bigvee_x \bigwedge_y G_{wxy}) \Rightarrow I_w$, there is an increase scope node S_w and a decrease scope node F_w. These nodes represent the side effects that a data structure update has on the model definition rules — in particular, that a data structure update may increase the scope of a model definition rule (i.e., cause the model definition rule to add a new object to a set or a new tuple to a relation) or decrease the scope of a model definition rule (i.e., cause the removal of an object from a set or a tuple from relation).
- **Consequence and compensation nodes:** For each model definition rule M_w, there is a pair of rule consequence nodes $\mathcal{C}_{wT}$ and $\mathcal{C}_{wF}$. The consequence nodes represent the consequences of increasing or decreasing the scope of a given model definition rule. For each model definition rule there is a set of compensation update nodes $\bigcup_z \{\mathcal{R}_{wz}\}$. The compensation update nodes represent data structure repairs that may be used to prevent the undesired scope increase of a model definition rule.

7.2 Edges in the Graph

The edges E in the graph represent various repair dependences.

- **Edges from model conjunction nodes to model repair nodes:** $\langle N_{ij}, A_{ijkl}\rangle \in E$ for each model conjunction node N_{ij} and each of the corresponding model repair nodes A_{ijkl}. These edges capture the dependences between model conjunctions and model repairs; there is an edge from a model conjunction node to the nodes corresponding to any of model repairs that may be required to satisfy the model conjunction.
- **Edges from model repair nodes:** $\langle A_{ijkl}, R_{ijklm}\rangle \in E$ for each model repair node A_{ijkl} and each of the data structure update nodes R_{ijklm} that implement the model repair. $\langle A_r, R_{rs}\rangle \in E$ for each model repair node A_r and all data structure update nodes R_{rs} that implement the model repair. $\langle A_{ijkl}, N_{i'j'}\rangle \in E$ (or $\langle A_r, N_{i'j'}\rangle \in E$) if the repair corresponding to A_{ijkl} (or A_r) may falsify the conjunction $C_{i'j'}$. $\langle A_{ijkl}, N_{i'j'}\rangle \in E$ (or $\langle A_r, N_{i'j'}\rangle \in E$) if the repair corresponding to A_{ijkl} (or A_r) may expand the quantifier scope of the constraint containing the conjunction $C_{i'j'}$. $\langle A_{ijkl}, S_w\rangle \in E$ (or $\langle A_r, S_w\rangle \in E$) if the repair corresponding to A_{ijkl} (or A_r) may increase the scope of the model definition rule M_w. $\langle A_{ijkl}, F_w\rangle \in E$ (or $\langle A_r, F_w\rangle \in E$) if the repair corresponding to A_{ijkl} (or A_r) may decrease the scope of the model definition rule M_w.
- **Edges from data structure update nodes:** $\langle R_{ijklm}, A_r\rangle \in E$ (or $\langle R_{rs}, A_r\rangle \in E$) if performing the data structure update represented by the data structure update node R_{ijklm} (or R_{rs}) may require that the repair algorithm also perform the repair represented by the model repair node A_r. $\langle R_{ijklm}, F_w\rangle \in E$ (or $\langle R_{rs}, F_w\rangle \in E$) if performing the data structure update represented by the data structure update node R_{ijklm} (or R_{rs}) may decrease the scope of the model definition rule M_w. $\langle R_{ijklm}, S_w\rangle \in E$ (or $\langle R_{rs}, S_w\rangle \in E$) if performing the data structure update represented by the data structure update node R_{ijklm} (or R_{rs}) may increase the scope of the model definition rule M_w.
- **Edges from scope increase or decrease nodes:** $\langle S_w, \mathcal{C}_{wT}\rangle \in E$ and $\langle F_w, \mathcal{C}_{wF}\rangle \in E$ for each model definition rule M_w. $\langle S_w, \mathcal{R}_{wz}\rangle \in E$ for each model definition rule M_w and each corresponding compensation node $\mathcal{R}_{wz}$. These edges link the scope increase or decrease node to the consequences of increasing or decreasing the scope of a model definition rule and the repairs that may be invoked to avoid inadvertently increasing the scope of a model definition rule.
- **Edges from compensation update nodes:** $\langle \mathcal{R}_{wz}, A_r\rangle \in E$ if performing the compensation update represented by the compensation update node $\mathcal{R}_{wz}$ may require the repair algorithm to also perform the repair represented by the model repair node A_r. $\langle \mathcal{R}_{wz}, F_{w'}\rangle \in E$ if performing the data structure update represented by the data structure update node $\mathcal{R}_{wz}$ may decrease the scope of the model definition rule $M_{w'}$. $\langle \mathcal{R}_{wz}, S_{w'}\rangle \in E$ if performing the data structure update represented by the data structure update node R_{wz} may increase the scope of the model definition rule $M_{w'}$.
- **Edges from consequence nodes:** $\langle \mathcal{C}_{wT}, N_{ij}\rangle \in E$ if increasing the scope of the model definition rule M_w may falsify the conjunction C_{ij} or expand the scope of its quantifier. $\langle \mathcal{C}_{wF}, N_{ij}\rangle \in E$ if decreasing the scope of the model definition rule M_w may falsify the conjunction C_{ij}. $\langle \mathcal{C}_{wT}, F_{w'}\rangle \in E$ (or $\langle \mathcal{C}_{wF}, F_{w'}\rangle \in E$) if increasing (or decreasing) the scope of the model definition rule M_w may decrease the scope of the model definition rule $M_{w'}$. $\langle \mathcal{C}_{wT}, S_{w'}\rangle \in E$ (or $\langle \mathcal{C}_{wF}, S_{w'}\rangle \in E$) if increasing (or decreasing) the scope of the model definition rule M_w may increase the scope of the model definition rule $M_{w'}$. These edges model the effects that increasing or decreasing the scope of a model definition rule may have on model constraints and on other model definition rules.

7.3 Schema of Nodes and Edges

Figure 13 presents a schema of the edges and nodes in the repair dependence graph. This schema summarizes the nodes presented in Section 7.1 and the edges presented in Section 7.2. The schema contains the following nodes:

- **Rectangular node labeled N_{ij}:** This class of nodes represents the option of satisfying the model constraint C_i by satisfying the model conjunction C_{ij}.
- **Elliptical node labeled A_{ijkl}/A_r:** This class of nodes represents the model repairs used to satisfy the basic propositions in the model constraints or to add an object (or tuple) to a set (or relation).
- **Circular node labeled R_{ijklm}/R_{rs}:** This class of nodes represents the data structure updates used to implement the model repairs.
- **Bold rectangular node labeled S_w/F_w:** This class of nodes represents the action of increasing (or decreasing) the scope of a model definition rule.
- **Bold elliptical node labeled $\mathcal{R}_{wz}$:** This class of nodes represents the compensation actions taken in response to undesired increases in the scope of a model definition rule.
- **Bold elliptical node labeled $\mathcal{C}_{wT}/\mathcal{C}_{wF}$:** This class of nodes represents the consequences of an increase (or decrease) in the scope of a model definition rule.

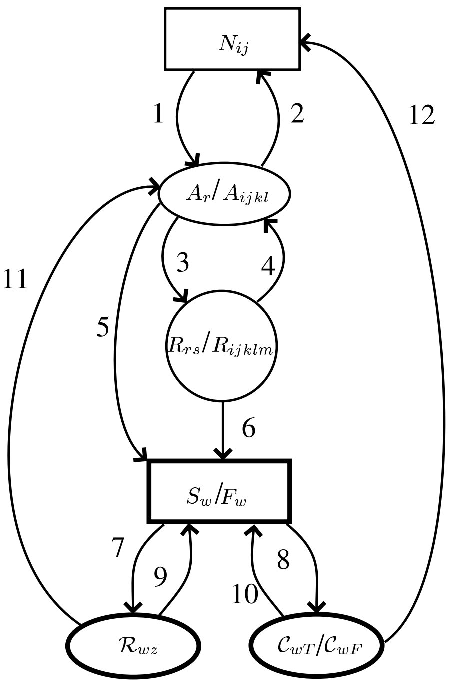

Figure 13: Repair Dependence Graph Schema

The edges in the schema represent the dependences between the nodes. The schema contains the following classes of edges:

1. There is an edge from a model conjunction node N_{ij} to a model repair node A_{ijkl} (or A_r) if the repair algorithm may need to perform the model repair corresponding to the model repair node A_{ijkl} (or A_r) to satisfy the conjunction C_{ij}.
2. There is an edge from a model repair node A_{ijkl} (or A_r) to a model conjunction node $N_{i'j'}$ if performing the model repair corresponding to the model repair node A_{ijkl} (or A_r) may falsify the conjunction $C_{i'j'}$.
3. There is an edge from a model repair node A_{ijkl} (or A_r) to a data structure update node R_{ijklm} (or R_{rs}) if the data structure update corresponding to the data structure update node R_{ijklm} (or R_{rs}) that may be performed to implement the model repair on the concrete data structures.
4. There is an edge from a data structure update node R_{ijklm} (or R_{rs}) to a model repair node A_r if part of performing the data structure update may require the repair algorithm to perform an additional model repair corresponding to the model repair node A_r.
5. There is an edge from a model repair node A_{ijkl} or A_r to a scope increase (or decrease) node S_w (or F_w) if performing the model repair corresponding to the model repair node may result in a scope increase (or decrease) of the model definition rule M_w. Note that these edges only capture dependences due to the abstract effects of a model repair.
6. There is an edge from a data structure update node R_{ijklm} (or R_{rs}) to a scope increase (or decrease) node S_w (or F_w) if performing the data structure update corresponding to the data structure update node may result in a scope increase (or decrease) of the model definition rule M_w. Note that these edges only capture dependences due to the concrete effects of a data structure update.
7. There is an edge from a scope increase node S_w to a compensation update node $\mathcal{R}_{wz}$ if the repair algorithm may perform the compensation updates corresponding to the compensation update node $\mathcal{R}_{wz}$ to compensate for an undesired increase in the scope of a the model definition rule M_w.
8. There is an edge from a scope increase (or decrease) S_w (or F_w) node to a consequence node $\mathcal{C}_{wT}$ (or $\mathcal{C}_{wF}$) if the repair algorithm does not perform a compensation update in response to undesired increases (or decreases) in the scope of N_w.
9. There is an edge from a compensation update node $\mathcal{R}_{wz}$ to a scope increase (or decrease) node $S_{w'}$ (or $F_{w'}$) if performing the compensation update may cause an increase (or decrease) in the scope of the model definition rule $M_{w'}$.
10. There is an edge from a consequence node $\mathcal{C}_{wT}$ (or $\mathcal{C}_{wF}$) to a scope increase node $S_{w'}$ if a consequence of a scope increase (or decrease) of the model definition rule M_w is a scope increase of the model definition rule $M_{w'}$. There is an edge from a consequence node $\mathcal{C}_{wT}$ (or $\mathcal{C}_{wF}$) to a scope decrease node $F_{w'}$ if a consequence of a scope increase (or decrease) of the model definition rule M_w is a scope decrease of the model definition rule $M_{w'}$.
11. There is an edge from a compensation update node $\mathcal{R}_{wz}$ to a model repair node A_r if part of performing

the compensation update corresponding to the compensation update node requires the repair algorithm to perform the model repair corresponding to the model repair node.

12. There is an edge from consequence node $\mathcal{C}_{wT}$ (or $\mathcal{C}_{wF}$) to a model conjunction node N_{ij} if a direct consequence of increasing (or decreasing) the scope of the model definition rule M_w is to falsify the conjunction C_{ij}.

7.4 Interference

To construct the graph, the algorithm must determine the dependences between repair actions, model constraints, and model definition rules. In particular, a model repair may change the scope of a model definition rule through the model definition rule's quantifier or through a guard of the form E `in` S or $\langle E, E\rangle$ `in` R in the model definition rule. A model repair may also falsify model constraints. Data structure and compensation updates may change the scope of a model definition rule by changing a data structure that the guard of the model definition rule accesses. Finally, a change in the scope of a model definition rule may effect model constraints or other model definition rules through their quantifiers or through their guards. We next discuss how the algorithm performs this dependence analysis.

7.4.1 Abstract Model Repair Actions

The foundation of the construction for determining interference due to model repairs is a procedure that determines if the repair of one basic proposition may *interfere* with a second basic proposition, i.e., if repairing the first proposition may falsify the second. Conceptually, the interference checking algorithm first checks if the two propositions involve disjoint parts of the model; if so, they do not interfere. If the two propositions may involve the same state, it reasons about the specific repair action and the second proposition. If the repair action is guaranteed to leave the model in a state that satisfies the second proposition, there is no interference. This is true if the first proposition implies the second. It may also be true even in some cases when the second proposition implies the first. For example, the two constraints $\texttt{size}(S) >= 1$ and $\texttt{size}(S) = 1$ do not interfere — the repair action for $\texttt{size}(S) >= 1$ makes $\texttt{size}(S) = 1$. The interference checking algorithm performs these checks using a table lookup to evaluate whether the repair of one conjunction may interfere with another. The table lookups rely on a few helper functions. We define the mapping $\mathcal{S}$ from a variable V to the set S that V quantifies over. We define the function $\phi(E, V)$ to be true iff the only variable references contained in E are to V. We define the function $\mathcal{NP}(S_1, S_2)$ to be true iff the partition constraints allow an element to be a member of both S_1 and S_2. The figures in Appendix B give the rules used to determine whether a model repair performed to satisfy a basic proposition will interfere with a second basic proposition.

Finally, model repairs can cause the scope of a model definition rule to increase or decrease. Repair actions that add an object (or tuple) to a set (or relation) may increase the scope of any model definition rule that quantifies over the set (or relation) or includes a non-negated guard that tests membership in the set (or relation). Repair actions that add an object (or tuple) to a set (or relation) may decrease the scope of any model definition rule that includes a negated guard that tests membership in the set (or relation). Repair actions that remove an object (or tuple) from a set (or relation) may decrease the scope of any model definition rule that quantifies over the set (or relation) or includes a non-negated guard that tests membership in the set (or relation). Repair actions that remove an object (or tuple) from a set (or relation) may increase the scope of any model definition rule that includes a negated guard that tests membership in the set (or relation).

7.4.2 Data Structure and Compensation Updates

Performing a data structure update or compensation update changes the concrete data structure. This change may cause additional increases or decreases in the scopes of the model definition rules.

- **Initial addition:** If an update can be determined to add the first element to a previously empty set, then the update does not decrease the scope of the model definition rule that was used to add objects to that set.

  Consider a model definition rule with an inclusion constraint of the form $\langle V, V.R\rangle$ with no quantifiers (or a quantifier and all of the rule's concrete propositions are of the form $V.\texttt{field} = E$, where V is a quantified variable). If an update can be determined to add the first object to the image of a given object under the relation, then the update does not cause any scope decreases for that model definition rule.

  Consider a model definition rule with an inclusion constraint of the form $\langle V.R, V\rangle$ with no quantifiers (or a quantifier and all of the rule's concrete propositions are of the form $V.\texttt{field} = E$, where V is a quantified variable). If an update can be determined to add the first object to the image of a given object under the inverse of the relation, then the update does not cause any scope decreases for that model definition rule.

- **Self propagation:** Consider an update intended to add an object to a set or a tuple to a relation. If the model definition rule used to generate the update contains a quantifier and the rule contains only concrete propositions of the form $V.\texttt{field} = E$ where V is a quantified variable, then the repair action does not further increase the scope of that model definition rule. If the model definition rule used to generate the update has no quantifiers, then the update does not further increase the scope of that model definition rule. A similar rule is true for decreasing the scope.

  Consider an update that performs an atomic modify operation. If the model definition rule used to generate the update contains a quantifier and it only contains concrete propositions of the form $V.\texttt{field} = E$ where V is the quantified variable, then the update does not further increase or decrease the scope of that model definition rule. If an update performs an atomic modify operation and the model definition rule used to generate the update does not contain a quantifier, then the update does not further increase or decrease the scope of that model definition rule.

- **Recursive data structures:** Consider an addition or removal update for a recursive data structure. If the model definition rule `for` V `in` $S, G' \Rightarrow FE'$ `in` S has a guard G' which contains only propositions of the form V.`field` $= E$ and the base case model definition rule does not quantify over any set, then the update does not increase or decrease the scope of this model definition rule or the base case model definition rule. Consider an addition or removal update for a recursive data structure. If the model definition rule `for` V `in` $S, G' \Rightarrow FE'$ `in` S has a guard G' which contains only propositions of the form V.`field` $= E$ and the base case model definition rule `for` V' `in` $S', G \Rightarrow FE$ `in` S has a guard G which contains only propositions of the form V'.`field` $= E$ then the update does not increase or decrease the scope of this model definition rule or the base case model definition rule.

- V.`field` $= V$ **may satisfy** !V.`field` $= NULL$**:** If an update satisfies the proposition V.`field` $= V$ where V is a quantification variable, then the update may decrease the scope of any model definition rule that contains the proposition V.`field` $= NULL$. The update may also increase the scope of any model definition rule that contains the proposition !V.`field` $= NULL$.

- V.`field` $= NULL$ **may satisfy** !V.`field` $= V$**:** If an update satisfies the proposition V.`field` $= NULL$, then the update may decrease the scope of any model definition rule that contains the proposition V.`field` $= V'$ and may increase the scope of any model definition rule that contains the proposition !V.`field` $= V'$.

- V.`field` $= E$ **may satisfy** V.`field` $= E$ **and may falsify** !V.`field` $= E$**:** If an update satisfies the proposition V.`field` $= E$ where E depends solely on V and globals, then the update may decrease the scope of any model definition rule that contains the proposition !V.`field` $= E$ and increase the scope of any model definition rule that contains the proposition V.`field` $= E$.

- **General case:** The general case is that setting V.`field` $= E$ may decrease or increase the scope of any model definition rules which use the field `field` (including in the inclusion constraint). This rule is only used by the repair algorithm if none of the previous rules apply.

7.4.3 Scope Increases and Decreases

Increases or decreases of a model definition rule's scope may change the abstract model. In particular, if the change in scope of a model definition rule causes an object (or tuple) to be added to or removed from a set (or relation), the resulting change in the model may falsify model constraints that depend on the set (or relation). As a result, the repair algorithm must perform further repairs to satisfy the violated constraints. The repair dependence graph must contain edges that account for this possibility. Appendix B gives the rules to determine whether an increase or decrease of the scope of a model definition rule may falsify a model constraint.

Finally, an increase or decrease in the scope of a rule may cause further increases or decreases in the scope of model definition rules. These further scope changes may falsify model constraints. As a result, the repair algorithm must perform further repairs to satisfy the violated constraints. The repair dependence graph must contain edges that account for these cascading scope changes. The graph must contain an edge between a scope increase or decrease node and a second scope increase or decrease node if the change in scope represented by the first node may cause the change in scope represented by the second node. The following are rules to determine whether a change in scope represented by the one scope increase or decrease node may cause a change in scope represented by the second scope increase or decrease node: Increases in the scope of a model definition rule that constructs a set (or relation) may increase the scope of any model definition rule that quantifies over the set (or relation) or includes a non-negated guard which test membership in the same set (or relation). Increases in the scope of a model definition rule constructs a set (or relation) may decrease the scope of any model definition rule that includes a negated guard which test membership in the same set (or relation). Decreases in the scope of a model definition rule that constructs a set (or relation) may decrease the scope of any model definition rule that quantifies over the set (or relation) or includes a non-negated guard which test membership in the same set (or relation). Decreases in the scope of a model definition rule that constructs a set (or relation) may increase the scope of any model definition rule that includes a negated guard which test membership in the same set (or relation).

7.5 Example Repair Dependence Graph

Figure 14 presents an example of a repair dependence graph generated by our implementation. This repair dependence graph corresponds to the model constraint `for p in ActiveProcesses, size(Next.p)<=1`. The rectangular nodes at the top of figure correspond to the different ways to satisfy the model constraint. The ellipses at the top of the figure correspond to the model repairs that satisfy the propositions in the rectangles. The circles correspond to the data structure updates that translate the model repairs to the concrete data structures. The bold rectangles correspond to scope decrease nodes for eighth and ninth model definition rules in Figure 4. The bold ellipses correspond to scope decrease consequence nodes. There are two cycles in the graph. The cycle involving a scope decrease node and a consequence node does not effect termination. The cycle involving the node labeled `list=null or  p in ActiveProcesses` is removed by the node pruning process described in Section 9. This example shows how the repair dependence graph captures the dependences in the repair process.

8. REPAIR ALGORITHM

The repair algorithm consists of the following steps:

1. **Initial Model Construction:** The repair algorithm initially constructs an abstract model as previously described in Section 4.

2. **Inconsistency Detection:** The repair algorithm evaluates the model constraints using the denotational semantics given in Figure 22 in the Appendix C. If the repair algorithm finds a violation of a model constraint it proceeds to the next step, otherwise the data structure is consistent and the repair process exits.

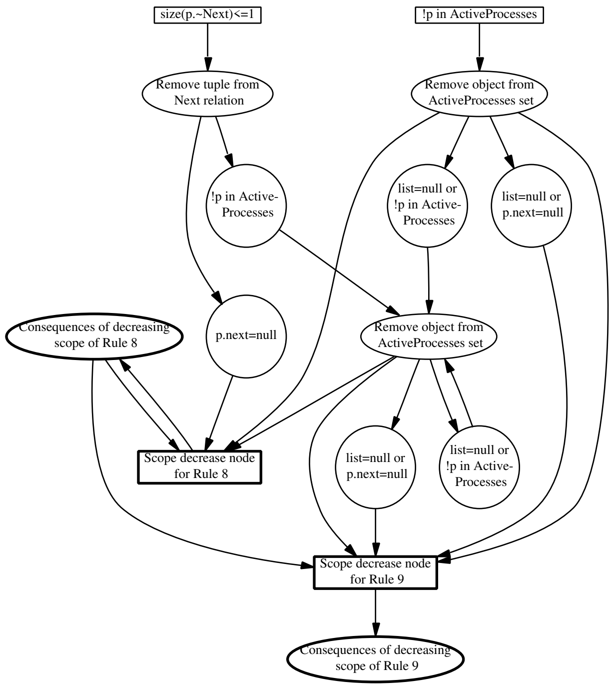

Figure 14: Example Repair Dependence Graph

3. **Conjunction Selection:** The repair algorithms computes the costs of each of the conjunctions that could be satisfied to repair a violated model constraint. [8] The cost of repairing a conjunction is the sum of the repair cost of each of the constituent basic propositions that must be satisfied. The repair algorithm chooses the least expensive conjunction to repair.

4. **Model Repair:** For each basic proposition that must be satisfied, the repair algorithm performs an abstract repair on the model. If an object (or a tuple) is added to a set (or a relation) or a tuple in a relation is modified, the repair algorithm immediately performs the corresponding data structure update. If an object (or tuple) is to be removed from a set (or a relation), the repair algorithm registers the data structure updates that remove the particular object. Object or tuple removal updates will be performed in Step 6 when the model is rebuilt. All other updates are performed in Step 5.

5. **Data Structure Updates:** Section 5.2 describes how the repair algorithm generates a set of data structure updates that implement a model repair. In some cases, a data structure update may require that an additional model repair be performed; in these cases, the repair algorithm performs Step 4 to perform the model repair.

6. **Model Update:** The repair algorithm performs the model construction described in Step 1. Whenever an object (or tuple) is added to a set (or relation), the repair algorithm checks if the object (or tuple) was in the set (or relation) in the previous version of the model from Step 4. If the object (or tuple) wasn't in the set (or relation), the repair algorithm first checks if a specific data structure update has been registered for the given object (or tuple) and set (or relation). If one has, the repair algorithm performs the given data structure update as described in Step 5. Otherwise it checks if a compensation update exists for the rule responsible for the addition of the new object (or tuple). If one exists, the repair algorithm performs the compensation update in the same manner as Step 5. If any data structure or compensation updates are performed, the model update is recomputed. Once the model update has completed, the repair algorithm deletes the old model and deletes all of the updates registered to a objects or tuples. Then the repair algorithm proceeds to Step 2.

[8]Note that the option to repair a given conjunction may be eliminated statically by the pruning performed by termination analysis given in Section 9 or by the developer.

9. TERMINATION

By construction, the edges in the graph capture all of the repair dependences of the repair algorithm. As a result, the transitive closure of the edges from a model conjunction node capture all of the possible effects of repairing that model conjunction. Any infinite repair therefore shows up as a cycle.

The repair dependence graph must be acyclic with the exception of cycles that solely contain scope decrease and consequence nodes, cycles that solely contain scope increase and consequence nodes, or cycles that are not reachable from the model conjunction nodes. [9] The repair algorithm may remove model conjunction nodes, data structure update nodes, and consequence/compensation update nodes to satisfy these cyclity constraints. The final graph must satisfy the following conditions in order to ensure that repairs exist for violated constraints:

1. There is at least one model conjunction node for each constraint in the model.

2. Each abstract repair node has at least one edge to a data structure update.

3. Each scope increase or decrease node has at least one edge to a consequence or compensation update node.

After modifying the graph, the algorithm never uses deleted repairs.

Theorem: If the graph for a given specification is acyclic with the exception of cycles that contain only scope decrease and consequence nodes, cycles that contain only scope increase and consequence nodes, or cycles that are not reachable from the model conjunction nodes and the graph satisfies the preceding three conditions, then the repair algorithm will terminate for the specification.

We provide a proof of this theorem in Appendix A.

[9]Note that these cycles do not effect termination as no work is associated with scope decrease cycles, scope increase cycles can only discover as many objects as exist in the heap, and the actions in unreachable cycles are never used.

10. RELATED WORK

We survey related work in software error detection [6, 7, 14, 4], traditional error recovery, manual data structure repair, and databases.

10.1 Traditional Error Recovery

Reboot potentially augmented with checkpointing is a traditional approach to error recovery. Database systems use a combination of logging and replay to avoid the state loss normally associated with rolling back to a previous checkpoint [11]. Transactions support consistent atomic operations by discarding partial updates if the transaction fails before committing. There has recently been renewed interest in applying many of these classical techniques in new computational environments such as Internet services [22] and in extending these techniques to reboot a minimal set of components rather than the complete system [1].

10.2 Manual Data Structure Repair

The Lucent 5ESS telephone switch [15, 13, 17, 12] and IBM MVS operating system [21] use inconsistency detection and repair to recover from software failures. The software in both of these systems contains a set of manually coded procedures that periodically inspect their data structures to find and repair inconsistencies. The reported results indicate an order of magnitude increase in the reliability of the system [11].

10.3 Constraint Programming

Researchers have incorporated constraint mechanisms into programming languages. One such system is Kaleidoscope [19]. Kaleidoscope allows the developer to specify constraints that the system should maintain. The developer is intended to write programs using a hybrid of imperative style programming and constraints where appropriate. Kaleidoscope does not include any analog of our model-based approach, as a result it can be very difficult if not impossible to express constraints on recursive data structures or other heap structures containing multiple elements. Another example of a constraint maintenance system as a programming abstraction is Alphonse [16]. Rule based programmer [20, 18] is a related technique in which the developer defines a test condition and an action to take in response.

10.4 Integrity Maintenance in Databases

Database researchers have developed integrity management systems that enforce database consistency constraints. These systems typically operate at the level of the tuples and relations in the database, not the lower-level data structures that the database uses to implement this abstraction. One approach is to provide a system that assists the developer in creating a set of production rules that maintain the integrity of a database [3]. This approach has been extended to enable the system to automatically generate both the triggering components and the repair actions [2]. Researchers in database view maintenance use incremental recomputation techniques to maintain view invariants [5]. Researchers have also developed a database repair system that enforces Horn clause constraints and schema constraints (which can constrain a relation to be a function) [23]. Our system supports a broader class of constraints — logical formulas instead of Horn clauses. It also supports constraints which relate the value of a field to an expression involving the size of a set or the size of an image of an object under a relation. Finally, it uses partition information to improve the precision of the termination analysis, enabling the verification of termination for a wider class of constraint systems.

10.5 Specification-Based Repair

In our previous research, we have developed a specification-based repair system that uses *external constraints* to explicitly translate the model repairs to the concrete data structures [9, 10, 8]. The primary disadvantage of this approach in comparison with the approach presented in this paper is a potential lack of repair effectiveness — there is no guarantee that the external constraints correctly implement the model repairs, and therefore no guarantee that the concrete data structures will be consistent after repair.

11. CONCLUSION

Data structure repair can be an effective technique for enabling programs to recover from data structure damage to continue to execute successfully. A developer using our model-based approach specifies how to translate the concrete data structures into an abstract model, then uses the sets and relations in the model to state key data structure consistency constraints. Our automatically generated repair algorithm finds and repairs any data structures that violate these properties. The key results in this paper include a technique for analyzing the model definition rules to translate model repairs into data structure updates and the use of the repair dependence graph to formulate and solve the repair termination analysis problem. This approach promises to substantially reduce the development costs and increase the effectiveness of data structure repair, enabling its application to a wider range of software systems.

11.1 Acknowledgments

We would like to thank Michael Ernst for useful feedback and discussions concerning this paper.

12. REFERENCES

[1] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany*, May 2001.

[2] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems*, 19(3), September 1994.

[3] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of 1990 VLDB Conference*, pages 566–577.

[4] J.-D. Choi and et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, 2002.

[5] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD Conference 1996*, pages 469–480, 1996.

[6] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[7] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, 2002.

[8] B. Demsky and M. Rinard. Automatic data structure repair for self-healing systems. In *First Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[9] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.

[10] B. Demsky and M. Rinard. Static specification analysis for termination of specification-based data structure repair. In *14th.IEEE International Symposium on Software Reliability Engineering*, November 2003.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[12] T. Griffin, H. Trickey, and C. Tuckey. Generating update constraints from PRL5.0 specifications. In *Preliminary report presented at AT&T Database Day*, September 1992.

[13] N. Gupta, L. Jagadeesan, E. Koutsofios, and D. Weiss. Auditdraw: Generating audits the FAST way. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, 1997.

[14] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, 2002.

[15] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.

[16] R. Hoover. Incremental computation as a programming abstraction. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, 1992.

[17] D. A. Ladd and J. C. Ramming. Two application languages in software production. In *Proceedings of the 1994 USENIX Symposium on Very High Level Language*, October 1994.

[18] D. Litman, A. Mishra, and P. Patel-Schneider. Modeling dynamic collections of interdependent objects using path-based rules. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1997.

[19] G. Lopez. *The Design and Implementation of Kaleidoscope, A Constraint Imperative Programming Language*. PhD thesis, University of Washington, April 1997.

[20] A. Mishra, J. Ros, A. Singhal, G. Weiss, D. Litman, P. Patel-Schneider, D. Dvorak, and J. Crawford. R++: Using rules in object-oriented designs. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, July 1996.

[21] S. Mourad and D. Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, September 1987.

[22] D. A. Patterson and et al. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, March 15, 2002.

[23] S. D. Urban and L. M. Delcambre. Constraint analysis: A design process for specifying operations on objects. *IEEE Transactions on Knowledge and Data Engineering*, 2(4), December 1990.

APPENDIX

A. TERMINATION PROOF

We separate the proof of termination into two parts: the first part shows that the concrete implementation of each abstract repair action terminates, and the second part shows that the repair of the abstract model using these abstract repair actions terminates.

A.1 Individual Abstract Repairs Terminate

Since an abstract repair action is completed before starting the next abstract repair action, the repair algorithm can only perform repair actions which are reachable in the repair graph from the abstract repair node without traversing a model conjunction node.

This reachable subgraph contains the following classes of nodes: Abstract repair action nodes, data structure update nodes, rule side effect nodes, consequence nodes, and compensation update nodes. As a result of the restrictions on cycles in the complete graph, strongly connected components in this subgraph can consist of: a single abstract repair action node, a single data structure update node, a single compensation update node, a set of scope increase and consequence nodes, or a set of scope decrease and consequence nodes. After the initial data structure updates(which trivially terminate), the repair algorithm only performs operations for the compensation update nodes.

It is clear that individual abstract repair actions terminate. Consider the strongly connected components of the subgraph in topologically sorted order. Once the first compensation update node is used to prevent all of the initial undesired increase in the scope of that rule, no further scope increases due to that rule will occur (otherwise there would be an incoming edge to this node). Once the first rule node has been repaired, no further activations of rules corresponding to the next rule node will occur, and so forth.

Notice that the fact that individual abstract repairs terminate implies that any infinite repair chains must involve infinitely many model conjunction repairs.

A.2 Repair Termination

Now that we've shown that individual abstract repairs terminate, we construct a new graph that summarizes only the dependences between model conjunction nodes. This graph is the transitive closure of the repair dependence graph restricted to the model conjunction nodes. By construction, there is an edge between two nodes if and only if repairing the first conjunction may falsify the second conjunction. The absence of undesirable cycles in the repair dependency graph ensures that this new graph has no cycles. We now show by structural induction that the absence of cycles in this new graph ensures that all repairs terminate.

Proof: (Structural induction).

(Base Case:) The base case, an acyclic graph of size 0, terminates because there are no violated conjunctions.

(Induction Step:) We assume that repairs terminate on all acyclic graphs of size k or less. We must show that all repairs terminate for an acyclic graph of size $k+1$.

Since the graph is acyclic, it must contain a node n with no incoming edges. Furthermore, all nodes corresponding to the same model constraint have no incoming edges arising from a possible quantifier scope expansion. Otherwise the node n would have a similar incoming edge as it shares the same quantifiers with the other nodes from the same constraint. Because there are no incoming edges to node n, the algorithm repairs each quantifier binding for n at most once — once the node is satisfied for a given quantifier binding, no other repair will falsify it. Therefore, the conjunction represented by node n may only be repaired a number of times equal to the number of quantifier bindings for the constraint that the conjunction appears in.

By the induction hypothesis, repairs on acyclic graphs of size k terminate. So after each repair of node n the algorithm either eventually repairs all violations of conjunctions corresponding to the other k nodes (leaving only violations of the conjunction corresponding to node n to possibly repair) or it repairs a violation of the node n before finishing the repairs on the other nodes. Since the conjunction represented by node n may only be repaired a number of times equal to the number of quantifier bindings for the constraint the conjunction appears in, the repair must eventually terminate.

B. ABSTRACT INTERFERENCE

In all of the figures in this section, we omit any cases in which a repair action never interferes with a given basic proposition.

Addition to set S_1 to satisfy $\mathtt{size}(S_1)\mathtt{=}c$, $\mathtt{size}(S_1)\mathtt{>=}c$, $\mathtt{!size}(S_1)\mathtt{=}c-1$, or $\mathtt{!size}(S_1)\mathtt{<=}c-1$.

$\mathcal{IF}(\mathtt{size}(S_2)\mathtt{=}c')$ $(S_1 = S_2) \wedge (c' < c)$
$\mathcal{IF}(\mathtt{size}(S_2)\mathtt{<=}c')$ $(S_1 = S_2) \wedge (c' < c)$
$\mathcal{IF}(\mathtt{!size}(S_2)\mathtt{=}c')$ $(S_1 = S_2) \wedge (c' = c)$
$\mathcal{IF}(\mathtt{!size}(S_2)\mathtt{>=}c')$ $(S_1 = S_2) \wedge (c' \leq c)$
$\mathcal{IF}(\mathtt{!}V\ \mathtt{in}\ S_2)$ $S_1 = S_2$

Addition to set S_1 to satisfy $V\ \mathtt{in}\ S_1$.

$\mathcal{IF}(\mathtt{size}(S_2)\mathtt{=}c')$ $(S_1 = S_2)$
$\mathcal{IF}(\mathtt{size}(S_2)\mathtt{<=}c')$ $(S_1 = S_2)$
$\mathcal{IF}(\mathtt{!size}(S_2)\mathtt{=}c')$ $(S_1 = S_2)$
$\mathcal{IF}(\mathtt{!size}(S_2)\mathtt{>=}c')$ $(S_1 = S_2)$
$\mathcal{IF}(\mathtt{!}V\ \mathtt{in}\ S_2)$ $S_1 = S_2$

Removal from set S_1 to satisfy $\mathtt{size}(S_1)\mathtt{=}c$, $\mathtt{size}(S_1)\mathtt{<=}c$, $\mathtt{!size}(S_1)\mathtt{=}c+1$, or $\mathtt{!size}(S_1)\mathtt{>=}c+1$.

$\mathcal{IF}(\mathtt{size}(S_2)\mathtt{=}c')$ $(S_1 = S_2) \wedge (c' > c)$
$\mathcal{IF}(\mathtt{size}(S_2)\mathtt{>=}c')$ $(S_1 = S_2) \wedge (c' > c)$
$\mathcal{IF}(\mathtt{!size}(S_2)\mathtt{=}c')$ $(S_1 = S_2) \wedge (c' = c)$
$\mathcal{IF}(\mathtt{!size}(S_2)\mathtt{<=}c')$ $(S_1 = S_2) \wedge (c' \geq c)$
$\mathcal{IF}(V\ \mathtt{in}\ S_2)$ $S_1 = S_2$

Removal from set S_1 to satisfy $\mathtt{!}V\ \mathtt{in}\ S_1$.

$\mathcal{IF}(\mathtt{size}(S_2)\mathtt{=}c')$ $(S_1 = S_2)$
$\mathcal{IF}(\mathtt{size}(S_2)\mathtt{>=}c')$ $(S_1 = S_2)$
$\mathcal{IF}(\mathtt{!size}(S_2)\mathtt{=}c')$ $(S_1 = S_2)$
$\mathcal{IF}(\mathtt{!size}(S_2)\mathtt{<=}c')$ $(S_1 = S_2)$
$\mathcal{IF}(V\ \mathtt{in}\ S_2)$ $S_1 = S_2$

Figure 15: Rules for computing interference from set additions and removals

Increase in the scope of a model definition rule that constructs set S_1.

$\mathcal{IF}(\texttt{size}(S_2)\texttt{=}c')$	$(S_1 = S_2)$
$\mathcal{IF}(\texttt{size}(S_2)\texttt{<=}c')$	$(S_1 = S_2)$
$\mathcal{IF}(\texttt{!size}(S_2)\texttt{=}c')$	$(S_1 = S_2)$
$\mathcal{IF}(\texttt{!size}(S_2)\texttt{>=}c')$	$(S_1 = S_2)$
$\mathcal{IF}(\texttt{!}V \texttt{ in } S_2)$	$S_1 = S_2$

Decrease in the scope of a model definition rule that constructs set S_1.

$\mathcal{IF}(\texttt{size}(S_2)\texttt{=}c')$	$(S_1 = S_2)$
$\mathcal{IF}(\texttt{size}(S_2)\texttt{>=}c')$	$(S_1 = S_2)$
$\mathcal{IF}(\texttt{!size}(S_2)\texttt{=}c')$	$(S_1 = S_2)$
$\mathcal{IF}(\texttt{!size}(S_2)\texttt{<=}c')$	$(S_1 = S_2)$
$\mathcal{IF}(V \texttt{ in } S_2)$	$S_1 = S_2$

Figure 16: Rules for computing interference from model definition rule scope changes

Addition to a relation to satisfy $\texttt{size}(V_1.R_1)\texttt{=}c$, $\texttt{size}(V_1.R_1)\texttt{>=}c$, $\texttt{!size}(V_1.R_1)\texttt{=}c-1$, or $\texttt{!size}(V_1.R_1)\texttt{<=}c-1$

$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' < c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge (c' < c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' = c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge (c' \leq c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2)$
$\mathcal{IF}(V_2.R_2 \textit{ comp } E))$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(V_2.R'.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{R}(R'))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(VE.R'.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{R}(R'))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3))$

Addition to relation to satisfy $V' \texttt{ in } V_1.R_1$

$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2)) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_3)$
$\mathcal{IF}(V_2.R_2 \textit{ comp } E))$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(V_2.R'.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{R}(R'))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(VE.R'.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{R}(R'))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))$

Figure 17: Rules for computing interference from relation additions

Removal from a relation to satisfy $\texttt{size}(V_1.R_1)\texttt{=}c$, $\texttt{size}(V_1.R_1)\texttt{<=}c$, $\texttt{!size}(V_1.R_1)\texttt{=}c+1$, or $\texttt{!size}(V_1.R_1)\texttt{>=}c+1$

$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' > c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge (c' > c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(V_3 \texttt{ in } V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' = c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge (c' \geq c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(V_2.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(V_2.R'.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{R}(R'))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(VE.R'.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{R}(R'))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(V_3 \texttt{ in } R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3))$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$

Removal from a relation to satisfy $\texttt{!}V' \texttt{ in } V_1.R_1$.

$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(V_3 \texttt{ in } V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(V_2.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(V_2.R'.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{R}(R'))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(VE.R'.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{R}(R'))) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))$
$\mathcal{IF}(V_3 \texttt{ in } R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3))$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))$

Figure 18: Rules for computing interference from relation removals

Modification to a relation to satisfy $V_1.R_1\ comp_1\ E_1$

$\mathcal{IF}(V_3 \texttt{ in } V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(V_2.R_2\ comp_2\ E_2)$	$(((comp_1 \neq comp_2) \vee (E_1 \neq E_2[V_2/V_1]) \vee \phi(E_1, V_1) \vee \phi(E_2, V_2)) \wedge (R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))) \vee (E_2 \texttt{ uses } R_1)$
$\mathcal{IF}(V_2.R'.R_2\ comp_2\ E_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{R}(R'))) \vee (E_2 \texttt{ uses } R_1)$
$\mathcal{IF}(VE.R'.R_2\ comp_2\ E_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{R}(R'))) \vee (E_2 \texttt{ uses } R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(V_3 \texttt{ in } R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3))$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3))$

Modification to a relation to satisfy $V_1.R'_1...R'_n.R_1\ comp_1\ E_1$

$\mathcal{IF}(V_3 \texttt{ in } V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{R}(R'_n), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{R}(R'_n), \mathcal{S}(V_2))$
$\mathcal{IF}(V_2.R_2\ comp_2\ E_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{R}(R'_n), \mathcal{S}(V_2))) \vee (E_2 \texttt{ uses } R_1)$
$\mathcal{IF}(V_2.R''_1...R''_{n'}.R_2\ comp_2\ E_2)$	$(((comp_1 \neq comp_2) \vee (E_1 \neq E_2[V_2/V_1]) \vee \phi(E_1, V_1) \vee \phi(E_2, V_2) \vee (n' \neq n) \vee (R'_1 \neq R''_1) \vee ... \vee (R'_n \neq R''_n)) \wedge (R_1 = R_2) \wedge \mathcal{NP}(\mathcal{R}(R'_n), \mathcal{R}(R''_{n'}))) \vee (E_2 \texttt{ uses } R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(V_3 \texttt{ in } R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{R}(R'_n), \mathcal{S}(V_3))$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{R}(R'_n), \mathcal{S}(V_3))$

Figure 19: Rules for computing interference from relation modifications

Increase in the scope of a model definition rule that constructs the relation R_1.

$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } V_2.R_2)$	$(R_1 = R_2)$
$\mathcal{IF}(V_2.R_2\ comp\ E))$	$(R_1 = R_2) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(V_2.R'.R_2\ comp\ E)$	$(R_1 = R_2) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(VE.R'.R_2\ comp\ E)$	$(R_1 = R_2) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{<=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{>=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } R_2.V_2)$	$(R_1 = R_2)$

Decrease in the scope of a model definition rule that constructs the relation R_1.

$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(V_3 \texttt{ in } V_2.R_2)$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(V_2.R_2\ comp\ E)$	$(R_1 = R_2) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(V_2.R'.R_2\ comp\ E)$	$(R_1 = R_2) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(VE.R'.R_2\ comp\ E)$	$(R_1 = R_2) \vee (E \texttt{ uses } R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{>=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(V_3 \texttt{ in } R_2.V_2)$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$(R_1 = R_2)$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{<=}c')$	$(R_1 = R_2)$

Figure 20: Rules for computing interference from model definition rule scope changes

C. DENOTATIONAL SEMANTICS

$$
\begin{array}{rcl}
hv & \in & HeapValue = Bit \cup Byte \cup Short \cup Integer \cup Struct \\
h & \in & Heap = \mathcal{P}(Object \times Field \times HeapValue \cup \\
 & & Object \times Field \times \mathbb{N} \times HeapValue) \\
v & \in & Value = \mathbb{Z} \cup Boolean \cup string \cup Struct \\
l & \in & Local = Var \rightharpoonup Value \\
s & \in & Store = Value \times Value \cup Value \\
m & \in & Model = \mathcal{P}(Var \times Store) \\
\mathcal{R} & : & M \to Heap \to Local \to Model \to Model \\
\mathcal{E} & : & E \to Heap \to Local \to Model \to Value \\
\mathcal{G} & : & G \to Heap \to Local \to Model \to Boolean \\
\mathcal{I} & : & I \to Heap \to Local \to Model \to Model \\
\mathcal{SE} & : & FE \to Heap \to Local \to Model \to Value
\end{array}
$$

$\mathcal{R}[V\ \texttt{in}\ S, M]\ h\ l\ m\ = \bigcup_{v \in m(S)} \mathcal{R}[M]\ h\ l[V \mapsto v]\ m$
$\mathcal{R}[\langle V_1, V_2\rangle\ \texttt{in}\ R, M]\ h\ l\ m\ =$
$\quad \bigcup_{\langle v_1, v_2\rangle \in S[S]\ l\ e} \mathcal{R}[M]\ h\ l[V_1 \mapsto v_1][V_2 \mapsto v_2]\ m$
$\mathcal{R}[\texttt{for}\ V = E_1..E_2, M]\ h\ l\ m\ =$
$\quad \bigcup_{v=\mathcal{E}[E_1]lm}^{\mathcal{E}[E_2]lm} \mathcal{R}[M]\ h\ l[V \mapsto v]\ m$
$\mathcal{R}[G \Rightarrow I]\ h\ l\ m\ = \text{if } (\mathcal{G}[G]\ h\ l\ m) \text{ then } (\mathcal{I}[I]\ h\ l\ m) \text{ else } m$
$\mathcal{G}[G_1\ \texttt{and}\ G_2]\ h\ l\ m\ = (\mathcal{G}[G_1]\ h\ l\ m) \wedge (\mathcal{G}[G_2]\ h\ l\ m)$
$\mathcal{G}[G_1\ \texttt{or}\ G_2]\ h\ l\ m\ = (\mathcal{G}[G_1]\ h\ l\ m) \vee (\mathcal{G}[G_2]\ h\ l\ m)$
$\mathcal{G}[!G]\ h\ l\ m\ = \neg(\mathcal{G}[G]\ h\ l\ m)$
$\mathcal{G}[E_1\texttt{=}E_2]\ h\ l\ m = (\mathcal{E}[E_1]\ h\ l\ m) == (\mathcal{E}[E_2]\ h\ l\ m)$
$\mathcal{G}[E_1\texttt{<}E_2]\ h\ l\ m\ = (\mathcal{E}[E_1]\ h\ l\ m) < (\mathcal{E}[E_2]\ h\ l\ m)$
$\mathcal{G}[E_1\texttt{<=}E_2]\ h\ l\ m = (\mathcal{E}[E_1]\ h\ l\ m) \leq (\mathcal{E}[E_2]\ h\ l\ m)$
$\mathcal{G}[E_1\texttt{>=}E_2]\ h\ l\ m = (\mathcal{E}[E_1]\ h\ l\ m) \geq (\mathcal{E}[E_2]\ h\ l\ m)$
$\mathcal{G}[E_1\texttt{>}E_2]\ h\ l\ m\ = (\mathcal{E}[E_1]\ h\ l\ m) > (\mathcal{E}[E_2]\ h\ l\ m)$
$\mathcal{G}[\texttt{true}]\ h\ l\ m = true$
$\mathcal{G}[E\ \texttt{in}\ S]\ h\ l\ m\ = \langle S, \mathcal{E}[E]\ h\ l\ m\rangle \in m$
$\mathcal{G}[\langle E_1, E_2\rangle\ \texttt{in}\ R]\ h\ l\ m\ = \langle R, \langle \mathcal{E}[E_1]\ h\ l\ m, \mathcal{E}[E_2]\ h\ l\ m\rangle\rangle \in m$
$\mathcal{I}[FE\ \texttt{in}\ S]\ h\ l\ m\ = m \cup \{\langle S, \mathcal{SE}[FE]\ h\ l\ m\rangle\}$
$\mathcal{I}[\langle FE_1, FE_2\rangle\ \texttt{in}\ R]\ h\ l\ m\ =$
$\quad m \cup \{\langle R, \langle \mathcal{SE}[FE_1]\ h\ l\ m, \mathcal{SE}[FE_2]\ h\ l\ m\rangle\rangle\}$
$\mathcal{E}[FE]\ h\ l\ m\ = \mathcal{SE}[FE]\ h\ l\ m$
$\mathcal{E}[string]\ h\ l\ m\ = string$
$\mathcal{E}[number]\ h\ l\ m\ = number$
$\mathcal{E}[E_1 \oplus E_2]\ h\ l\ m\ = primop(\oplus, (\mathcal{E}[E_1]\ h\ l\ m), (\mathcal{E}[E_2]\ h\ l\ m))$
$\mathcal{SE}[V]\ h\ l\ m\ = l(V)$
$\mathcal{SE}[V.field]\ h\ l\ m\ = b.\langle l(V), field, b\rangle \in h$

Figure 21: Denotational Semantics for the Model Definition Language

$$
\begin{array}{rcl}
v & \in & Value = Number \cup Boolean \cup string \cup Object \\
l & \in & Local = \mathcal{P}(Var \times Value) \\
m & \in & Model = \mathcal{P}(Var \times Store) \\
s & \in & Store = Value \times Value \cup Value \\
\mathcal{EV} & : & C \to Local \to Model \to Boolean \\
\mathcal{E} & : & E \to Local \to Model \to Value \\
\mathcal{C} & : & B \to Local \to Model \to Boolean \\
\mathcal{V} & : & VE \to Local \to Model \to Value \\
\mathcal{PR} & : & P \to Local \to Model \to Boolean \\
\mathcal{SE} & : & SE \to Local \to Model \to \mathcal{P}(Value)
\end{array}
$$

$\mathcal{EV}[\texttt{for}\ V\ \texttt{in}\ S, C]\ l\ m\ =$
$\quad \bigwedge_{v \in m(S)} \mathcal{EV}[C]\ l[V \mapsto v]\ m$
$\mathcal{EV}[\texttt{for}\ \langle V_1, V_2\rangle\ \texttt{in}\ R, C]\ l\ m\ =$
$\quad \bigwedge_{\langle v_1, v_2\rangle \in m(R)} \mathcal{EV}[C]\ l[V_1 \mapsto v_1][V_2 \mapsto v_2]\ m$
$\mathcal{EV}[B]\ l\ m = \mathcal{C}[B]\ l\ m$
$\mathcal{C}[B_1\ \texttt{and}\ B_2]\ l\ m = \mathcal{C}[B_1]\ l\ m \wedge \mathcal{C}[B_2]\ l\ m$
$\mathcal{C}[B_1\ \texttt{or}\ B_2]\ l\ m\ = \mathcal{C}[B_1]\ l\ m \vee \mathcal{C}[B_2]\ l\ m$
$\mathcal{C}[!B]\ l\ m = \neg\mathcal{C}[B]\ l\ m$
$\mathcal{C}[P]\ l\ m\ = \mathcal{PR}[P]\ l\ m$
$\mathcal{PR}[VE\texttt{=}E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m == \mathcal{E}[E]\ l\ m)$
$\mathcal{PR}[VE\texttt{<}E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m < \mathcal{E}[E]\ l\ m)$
$\mathcal{PR}[VE\texttt{<=}E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m \leq \mathcal{E}[E]\ l\ m)$
$\mathcal{PR}[VE\texttt{>}E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m > \mathcal{E}[E]\ l\ m)$
$\mathcal{PR}[VE\texttt{>=}E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m \geq \mathcal{E}[E]\ l\ m)$
$\mathcal{PR}[V\ \texttt{in}\ SE]\ l\ m\ = l(V) \in \mathcal{SE}[SE]\ l\ m$
$\mathcal{PR}[\texttt{size}(SE)\texttt{=}c]\ l\ m\ = \mathcal{E}[\texttt{size}(SE)]\ l\ m\ == c$
$\mathcal{PR}[\texttt{size}(SE)\texttt{>=}c]\ l\ m\ = \mathcal{E}[\texttt{size}(SE)]\ l\ m\ \geq c$
$\mathcal{PR}[\texttt{size}(SE)\texttt{<=}c]\ l\ m\ = \mathcal{E}[\texttt{size}(SE)]\ l\ m\ \leq c$
$\mathcal{V}[V.R]\ l\ m\ = y.\langle l(V), y\rangle \in m(R)$
$\mathcal{V}[VE.R]\ l\ m\ = y.\langle \mathcal{V}[VE]\ l\ m, y\rangle \in m(R)$
$\mathcal{E}[V]\ l\ m\ = l(V)$
$\mathcal{E}[E_1 \oplus E_2]\ l\ m\ = primop(\oplus, \mathcal{E}[E_1]\ l\ m, \mathcal{E}[E_2]\ l\ m)$
$\mathcal{E}[E.R]\ l\ m\ = y.\exists z, z \in \mathcal{E}[E]\ l\ m \wedge \langle z, y\rangle \in m(R)$
$\mathcal{E}[\texttt{size}(SE)]\ l\ m\ =\mid \mathcal{SE}[SE]\ l\ m \mid$
$\mathcal{SE}[S]\ l\ m\ = \{s \mid s \in m(S)\}$
$\mathcal{SE}[V.R]\ l\ m\ = \{y \mid \langle l(V), y\rangle \in m(R)\}$
$\mathcal{SE}[VE.R]\ l\ m\ = \{y \mid \exists x.\langle x, y\rangle \in m(R) \wedge x \in \mathcal{SE}[VE]\ l\ m\}$
$\mathcal{SE}[R.V]\ l\ m\ = \{y \mid \langle y, l(V)\rangle \in m(R)\}$

Figure 22: Denotational Semantics for Model Constraints