Data Structure Repair Using Goal-Directed Reasoning

by

Brian C. Demsky

Submitted to the Electrical Engineering and Computer Science Department
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2006

Author .....
Electrical Engineering and Computer Science Department
January 19, 2006

Certified by .....
Martin C. Rinard
Associate Professor in Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Data Structure Repair Using Goal-Directed Reasoning

by

Brian C. Demsky

Submitted to the Electrical Engineering and Computer Science Department
on January 19, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Software errors, hardware faults, and user errors can cause data structures in running applications to become damaged so that they violate key consistency properties. As a result of this violation, the application may produce unacceptable results or even crash.

This dissertation presents a new data structure repair system that accepts a specification of key data structure consistency constraints, then generates repair algorithms that dynamically detect and repair violations of these constraints, enabling the application to continue to execute productively even in the face of otherwise crippling errors. We have successfully applied our system to five benchmarks: CTAS, an air traffic control tool; AbiWord, an open source word processing application; Freeciv, an online game; a parallel x86 emulator; and a simplified Linux file system. Our experience using our system indicates that the specifications are relatively easy to develop once one understands the data structures. Furthermore, for our set of benchmark applications, our experimental results show that our system can effectively repair inconsistent data structures and enable the application to continue to operate successfully.

Thesis Supervisor: Martin C. Rinard
Title: Associate Professor in Electrical Engineering and Computer Science

Acknowledgements

I owe a debt of gratitude to many people, for their assistance and guidance in my research . First, to my advisor, Martin Rinard, for without his guidance, patience, encouragement, wisdom, and help this project could not have been done. I would like to thank the remaining members of my committee, Michael Ernst and Daniel Jackson, for their time and contributions to this project. I would like to thank the Fannie and John Hertz Foundation for their fellowship.

I would like to thank Scott Ananian, Patrick Lam, Viktor Kuncak, Darko Marinov, Maria Cristina Marinescu, Radu Rugina, Alexandru Salcianu, Amy Williams, Karen Zee, and Mary McDavitt for their support and assistance.

I would like to thank Daniel Roy for his specification checking code and Philip Guo for his Dwarf2 code that was used to develop the automatic structure extraction tool.

I would like to thank all of the great friends I have made during my stay at MIT for making MIT enjoyable. I would like to thank Fuji Lai for being there this entire time. Finally, I would like to thank my parents and my brother for their moral support.

Contents

List of Figures

Chapter 1

Introduction

Applications often make assumptions about the states of the data structures that they manipulate. Errors that cause these data structures to become inconsistent and violate these assumptions may cause software systems to behave unacceptably or even fail catastrophically. A common recovery strategy for data structure inconsistencies is to reboot the system, thereby eliminating the inconsistencies when the system rebuilds the data structures from scratch. However, this strategy may fail: if inconsistencies appear in persistent data structures or are triggered by recurring problematic inputs, then the system may continue to be completely inoperable, even after rebooting.

Motivated in part by the drawbacks of rebooting, researchers have developed hand-coded data structure repair techniques to address data structure corruption errors. Several very successful deployed systems use data structure inconsistency detection and repair techniques to improve reliability in the face of errors. For example, the Lucent 5ESS switch and IBM MVS operating systems both use hand-coded audit and repair procedures to recover from data structure corruption errors [21, 29]. The reported results indicate an order of magnitude increase in the reliability of these systems [18]. Similar repair procedures exist for persistent data structures such as file systems and application files [5, 3].

While hand-coded repair algorithms are a useful way to enable software systems to recover from data structure corruption events, they are difficult to implement, as the developer must reason about the operational details of the repair process to

obtain a correct repair algorithm. Because of the context in which they are used, repair algorithms also present additional challenges that are not present in standard software:

- One challenge is that repair algorithms must repair completely unconstrained data structures. For example, software errors, hardware errors, or user errors can leave data structures in arbitrary states. This can be difficult for the developer, as he or she must write code that traverses, checks, and repairs a data structure while making absolutely no assumptions about the state of that data structure. For example, code to process a broken linked list must check for invalid pointers, cycles in the linked list, sharing between linked lists, and overlapping linked list elements in addition to implementation-specific errors.

- A second challenge is that it is difficult to test repair algorithms. In particular, testing is complicated by the following issues: 1) testing must ensure that the repair algorithm functions on completely unconstrained inputs, and 2) there are many ways the data structure can be broken. In particular, the testing process must ensure that the repair algorithms: 1) terminate for all possible input data structures, 2) do not crash on any input data structures, and 3) successfully repair any damaged data structures.

1.1 Specification-Based Data Structure Repair

To address these challenges, we have developed a specification-based approach to data structure repair. This dissertation presents our specification-based data structure repair technique and our experience using this technique to enable systems to recover from data structure corruption.

Instead of manually developing ad-hoc procedures to repair inconsistent data structures, our technique allows the developer to write a declarative specification of the data structure consistency properties and annotate the application to indicate where these specifications should hold. When possible, our specification compiler

then compiles this specification into C code that checks that the consistency properties hold. When the user runs the application, the application invokes the generated repair algorithm at the developer annotated points. When invoked, the repair algorithm checks for constraint violations, repairs any violations that it finds, and finally returns control to the application.

This specification-based approach has the following key advantages:

- It enables the developer to focus on the consistency constraints rather than on the low-level operational details of developing algorithms that detect and correct violations of these constraints. Because of the declarative nature of the constraints and the enormous number of operational details the developer must take into account when developing repair algorithms by hand, we believe that it is easier for developer to specify constraints than to develop repair algorithms by hand.

- The highly-defensive and unnatural programming style necessary to access arbitrarily broken data structures can be automatically generated by the compiler. Since the specification compiler automatically generates the traversals it can systematically insert the necessary checks at every access.

- The specification compiler is likely to be more rigorously tested than any given hand-coded repair algorithm because it will be used to generate repair algorithms for many different applications. The repair algorithms generated by the specification compiler are therefore more likely to be correct (and therefore more likely to correctly repair any inconsistent data structures) than hand-coded repair algorithms.

- The specification compiler automatically reasons about interactions between repair actions and consistency properties to ensure that the repaired data structure correctly satisfies all of the consistency properties specified by the developer. Compared with hand-coded repair algorithms, the specification compiler automatically reasons about the interactions between repair actions and consistency properties. To write repair algorithms, the developer must reason about

all possible interactions between the repair actions and the consistency properties the code enforces. If the developer of a hand-coded repair algorithm forgets to account for an interaction, the hand-coded repair algorithm could fail to terminate or not repair some violated consistency property.

- The generated repair algorithm is guaranteed to terminate.

Because of all of these reasons, we expect our specification-based approach to substantially simplify the process of developing repair algorithms. One beneficial result of this simplification is that developers may be able to successfully incorporate data structure repair into a larger body of software systems.

1.2 Experience

Our data structure repair technique produces data structures that satisfy the consistency properties. While these repaired data structures are not guaranteed to be the same data structures that a correct execution would have produced, they are heuristically close to the original broken data structures. While in practice this may enable applications to continue to execute successfully, it does not ensure any properties about the continued execution of the application. Furthermore, whether a repaired execution is acceptable to the user is a subjective question that can only be evaluated by considering the deviation from correct behavior and the context in which the application is intended to be used. As a result, we believe that the most reasonable way to evaluate whether data structure repair enables applications to recover from otherwise serious or even fatal errors is to perform an empirical evaluation. A key question that our evaluation focuses on is whether data structure repair, when given a specification that detects an inconsistency, effectively enables applications to recover from that inconsistency in an acceptable manner.

To perform this evaluation, we first developed a compiler-based implementation of our data structure repair technique. This implementation compiles data structure consistency specifications into C code that implements a repair algorithm for the data

structure.

We then acquired five different applications and applied our technique to each. The five applications were: a word processor, a parallel x86 emulator, an air-traffic control system, a Linux file system, and an interactive game. For each of the applications we developed a workload that exercised the application and identified a pre-existing error or injected a fault to corrupt a data structure. We then wrote a specification that was targeted towards the particular constraint. Without data structure repair, the corrupted data structures caused each of the applications to crash. With data structure repair, the corruption was repaired and the applications were able to continue to execute successfully. In some cases, data structure repair leveraged the presence of redundant information to produce a completely correct data structure. In other cases, the repair produced data structures that potentially deviated from the correct data structure. Nevertheless, these repaired data structures satisfied the basic data structure consistency constraints and enabled the application to continue to execute acceptably.

To incorporate data structure repair in these applications, we had to write data structure specifications for each application. To simplify the evaluation, we first found a software error, then wrote the data structure consistency specification to cover the properties violated by the selected error. In this evaluation, we found that the specifications were relatively to easy to develop once we understood the consistency properties for the data structure and that most of the time we spent developing specifications was spent understanding the consistency properties of the data structures. Our evaluations leaves open the question of how hard it is to obtain comprehensive enough specifications to repair previously unknown errors and to ensure that repairs of one constraint do not violate other important but unstated consistency constraints..

1.3 Scope of Data Structure Repair

We do not expect that specification-based data structure repair will be suitable for all applications. The scope of data structure repair is limited by the acceptability of the

repaired data structures for each application in the context in which it is used and whether the application has properties that make data structure repair attractive.

1.3.1 Acceptability of Repaired Data Structures

A significant issue is that data structure repair does not necessarily generate the same data structure that a correct execution would have. As a result, the continued execution of the application after repair may diverge significantly from the correct execution and even produce unacceptable results. Our results, however, indicate that repairing the inconsistencies in the data structures almost always enables our applications to execute in an manner that we believe the user will find acceptable. Moreover, for these applications continued acceptable execution is more desirable than crashing and losing all of their volatile state. However, we recognize that there are applications where absolute correctness is more important than continued execution, and data structure repair is not appropriate for these applications. Examples of such computations include cryptography algorithms (which cannot tolerate even the slightest deviation from correct behavior) and many numerical calculations (for which confidence in accuracy of the output value is important and small deviations in the calculations can produce large differences in the output).

1.3.2 Target Applications

Our target applications have several properties that make data structure repair attractive:

- **Rebooting Ineffective:** There must be some reason why a user would not just reboot these applications to recover from inconsistencies. Some reasons that a user may wish to not simply reboot an application include: the application maintains persistent data that may become corrupt (such inconsistencies persist across reboots and may cause the rebooted system to crash), the system has important information in volatile data structures, the system may take too long to boot, the boot process may be complicated and involve human inter-

vention, or recurring problematic inputs may cause the system to repeatedly crash. Applications that maintain persistent data structures are particularly good candidates for data structure repair, because inconsistencies in these persistent data structures cannot be fixed by rebooting these applications.

- **Well-specified Data Structures:** The application should have well-specified data structures with important consistency properties. It can be difficult to write a useful consistency specification if the consistency properties are not known, if all of the possible data structure configurations are consistent, or if the consistency properties relate local variables in different naming contexts.

- **Tolerates Deviations from Complete Correctness:** The application must tolerate some deviation from complete correctness. Several different factors can help applications tolerate deviations from completeness correctness: 1) the application may have many decoupled subcomputations, some of which are not affected by the deviation; 2) the application may contain many independent objects only some of which are affected by the inconsistency and, therefore, the application is able to correctly process the correct objects; 3) the user may be able to tolerate large deviations from correct behavior; 4) the application may flush incorrect state and, therefore, eventually return to completely correct behavior; and 5) the consistency properties may be sufficient to ensure that the future behavior of the application is acceptable.

- **Application's Uptime is Important:** There must be a reason why the developers care enough about the application's uptime to apply data structure repair. This can either be that the application is important and/or widely deployed.

1.4 Specification Language

Our specification language uses a set- and relation-based abstract model of data structures. This model consists of sets of objects and relations between these objects. The developer writes consistency constraints in terms of these sets and relations.

Many of the objects in an application's data structures have identical consistency properties. For example, all of the nodes in a linked list should be pointed to by at most one other object in the list. We included sets in our abstract model because they provide a natural way to group together all of these objects so that the developer can then write a consistency property that holds for all objects in the set. The fields in objects associate the object with primitive values or other objects. Other data structures such as a hashtable can also associate an object with primitive values or other objects. We included relations in our abstract model because they provide a natural way to abstract both of these associations.

As we previously discussed, it is often useful to write constraints that must be true for all objects in a set. We have included guarded universal quantifiers to enable the developer to write a constraint that holds for all objects in a set or all tuples in a relation. Our specification language includes a set of basic propositions, which can be combined by standard logical connectives (or, and, negation). We chose the set of basic propositions so that they were easy to repair, and that they could express important consistency properties. These basic propositions include numerical inequalities (for example, a lower bound on an object's fields), pointer equalities (for example, constraints that ensure that back-pointers in a doubly-linked list are consistent with the forward pointers), set and relation inclusion constraints (for example, constraints that relate set membership to other properties), constraints on the sizes of sets (for example, constraints that ensure that data structures exist), and constraints on the in-degree and out-degree of relations with respect to a given object (for example, constraints that ensure that an object in a linked-list is referenced at most once by other objects in the list).

A key challenge in this approach is constructing the sets and relations in the abstract model from the concrete data structures. Our specifications include a set of developer-provided model definition rules, which the developer uses to specify the connection between the concrete data structures and the abstract model. The repair algorithm then uses these model definition rules to construct an abstract model of the concrete data structure.

We generate repair algorithms that use the model definition rules to traverse the concrete data structures in the heap to classify objects into sets and to construct the relations. The repair algorithm then checks that the consistency properties hold on the abstract model. If it finds a constraint violation, it selects a conjunction in the disjunctive normal form (DNF) of the constraint to make true. Note that if the selected conjunction in the constraint is true, the entire constraint is true. The repair algorithm then makes the conjunction true by performing model repairs to make each of the individual basic propositions that comprise the conjunction true. These model repairs make the basic propositions true by either adding or removing objects (or tuples) to or from sets (or relations). At this point the repair algorithm has repaired the abstract model, but it has not yet repaired the underlying data structure. The repair algorithm next uses goal-directed reasoning to analyze the model definition rules to translate these model repairs into data structure updates. After performing the concrete data structure update, the algorithm has repaired a single constraint violation in the underlying data structure. However, multiple constraints may be violated, so the the repair algorithm repeats the process until it finds no further violations. The end result of the repair process is that all of constraints hold on the repaired data structure.

1.5 Termination

In general, it is possible for a repair algorithm to become trapped in an infinite repair loop. This can happen, for example, if the repair algorithm repairs one constraint and in the process violates a second constraint, then repairs the second constraint and in the process violates the original constraint. The specification compiler determines if this kind of non-termination can happen by building the *repair dependence graph*, which tracks all of the possible dependences between repair actions. The absence of certain kinds of cycles in the repair dependence graph guarantees that the repair algorithm will always terminate. Even if this graph contains one or more of these cycles, the algorithm may in some cases be able to prune sets of nodes to eliminate

these cycles. This pruning corresponds to eliminating potential repair actions that the repair algorithm could otherwise use. Before the specification compiler generates the repair algorithm, it checks for the absence of any such cycles in the repair dependence graph. The generated repair algorithm will therefore always terminate and leave the data structure in a consistent state.

1.6 Contributions

This dissertation makes the following contributions:

- **Specification-Based Data Structure Repair:** It presents a specification-based approach to data structure repair. The developer simply writes the key consistency properties, and our tool automatically generates the corresponding data structure repair algorithm. This approach allows the developer to focus on the consistency properties and frees the developer from reasoning about the operational details of the repair algorithms.

- **Abstract Model:** Our approach is based on a set- and relation-based abstract model of the data structure. The developer uses this abstraction of the data structure to express key data structure consistency properties.

- **Specification language suitable for repair:** It presents a specification language that can express many data structure consistency properties and that can be used to automatically generate data structure repair algorithms. This specification language also specifies how to construct the set- and relation-based abstraction of the concrete data structures.

- **Model Construction:** It presents how the repair algorithm evaluates the model definition rules to construct a set- and relation-based abstract model of a concrete data structure.

- **Consistency Checking and Repair:** It presents how our repair algorithm checks the consistency properties on the abstract model and repairs any violated

constraints that it finds.

- **Repair Translation:** A key challenge with this approach is translating the repair actions from this abstraction into operations that update the concrete data structures. This dissertation presents an algorithm that uses goal-directed reasoning to translate repair actions into operations that update the concrete data structures.

- **Termination Analysis:** It presents a conservative analysis that determines whether the generated repair algorithm terminates. This analysis reasons about the interactions between the consistency constraints, the repair actions for these constraints, and the model definition rules to ensure that the generated repair algorithm always terminates.

- **Experience:** It presents our experience using data structure repair on several applications. We have used data structure repair to successfully enable five applications to recover from otherwise fatal data structure consistency corruption.

At this point we have completed an initial investigation of our technique and obtained positive results. However, several questions remain unanswered. How hard is it to obtain specifications that have good coverage of the properties that software errors will violate? How well will data structure repair work on a broader class of applications with (presumably) a broader class of errors? The next step would be to perform an evaluation in which we first develop a consistency specifications for a broader class of applications, find errors in the applications, and finally test whether the repair algorithm enables the applications to acceptably recover from these errors.

1.7 Organization

The remainder of the dissertation is organized as follows. Chapter 2 presents an overview of the repair technique. Chapter 3 presents a file system example. Chapter 4 presents the specification language in detail. Chapter 5 presents the general

repair strategy the generated repair algorithms utilize. Chapter 6 presents the repair dependence graph and describes how the algorithm uses this graph to ensure that the generated repair algorithms terminate. Chapter 7 presents a method for computing upper bounds on the time taken by a generated repair algorithm. Chapter 8 presents the mechanisms that the developer can use to control the generated repair algorithms. Chapter 9 presents our experience applying data structure repair to five benchmark applications. Chapter 10 presents related work. Chapter 11 presents our conclusions.

Chapter 2

Overview

Our repair algorithm starts by building an abstract relational model of the system's data structures. This abstract relational model is composed of sets of objects and relations between these objects. After the model is constructed, the repair algorithm checks the model consistency constraints on the abstract model. If the checking algorithm finds a violated consistency constraint, it begins the repair process. We implement the repair process as a sequence of insertions and removals of tuples from this relational model. Note that it is possible to perform any desired change to the model using a sequence of insertions and removals of tuples. A key issue with this approach is translating repairs made to the relational model back into changes to the bits. Our approach uses goal-directed reasoning to translate the repair actions for the abstract model into updates to the concrete data structures.

Figure 2-1 presents a graphical overview of the repair process. The boxes at the bottom of the figure represent the bits in the heap, and the rounded boxes at the top of the figure represent the relational model. The repair process performs the following steps:

1. **Structure Decoding:** This step in the repair process decodes the bit patterns in memory into objects. The developer specifies the layout of the objects using our structure definition language.

2. **Constructing Relational Model:** This step of the repair process finds the

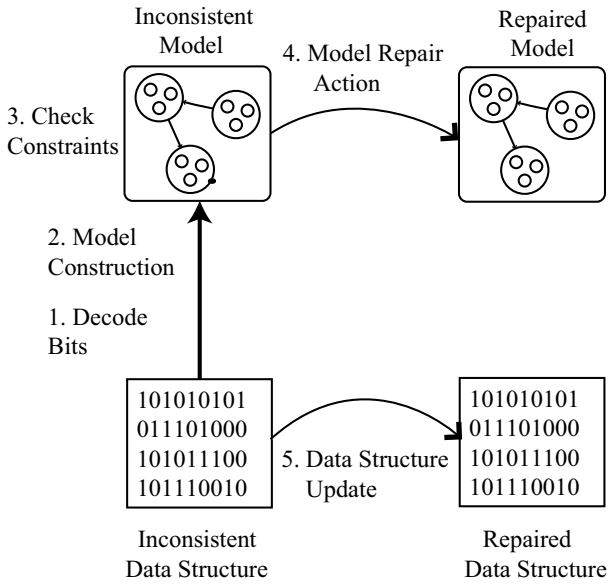


Figure 2-1: Repair Process

objects, places the objects into the appropriate sets, and constructs the relations in the model. It constructs the sets and relations by starting from a set of roots and tracing out the pointers in the heap. A set of model definition rules tells the repair algorithm how to classify the objects and construct the relations.

3. **Checking Properties:** This step checks for inconsistencies in the abstract model. The developer expresses the data structure consistency properties as a set of consistency constraints. Each constraint is a boolean combination of basic propositions involving the sets and relations in the model. This step evaluates each of these consistency constraints on the relational model.

4. **Perform Model Repair:** This step repairs any inconsistencies that the repair algorithm discovered in the previous step. To generate a repair for a violated constraint, the repair algorithm rewrites the constraint in disjunctive normal form (DNF), chooses one of the conjunctions in the DNF form of the constraint, and then applies model repairs as appropriate to make each of the individual basic propositions in the selected conjunction true. The repair algorithm implements these model repairs using a sequence of operations that either add or

remove objects (or tuples) to or from sets (or relations).

5. **Translating the Model Repairs:** This step translates the model repairs into data structure updates. As mentioned above, the repair algorithm uses a sequence of operations that either add or remove an object (or tuple) to or from a set (or relation) to implement each repair. To implement such a repair, the repair algorithm finds the model definition rules that construct a given set or relation, then manipulates the data structures to cause these model definition rules to either add or remove the object (or tuple) to or from the set (or relation). While we could generate data structure updates dynamically, our system statically pre-computes all possible model repairs and then for each possible model repair generates code that, when invoked, performs the corresponding data structure update. The repair algorithm then implements the data structure update by invoking the correct piece of pre-generated code.

6. **Repeat the Process:** The process discussed above repairs a single violation. To repair all of the constraint violations, the repair algorithm repeats the process until all of the constraints hold.

We next discuss each of the components of the repair algorithm in more detail.

2.1 Set and Relation Declarations

The developer writes set and relation declarations to declare the sets and relations that comprise the abstract relational model. A set declaration declares the name of the set and the type of objects or primitive values that this set contains. The set declaration can also declare that a set is a subset of another set, or that a set is partitioned into other sets. A relation declaration declares the name of the relation and relation's domain and range. The domain and range can be declared as either a set, a type of object, or a type of primitive value.

Figure 2-2 presents set and relation declarations for an example. The first line of this figure declares the set `AllInts` to contain objects of the type `IntObj`. Note that

```
set AllInts of IntObj
set NonNegInts of IntObj
set IntValues of integers
```

Figure 2-2: Example Set and Relation Declarations

Figure 2-3 in Section 2.2.1 presents the definition for the `IntObj` type. The second line declares the set `NonNegInts` to contain objects of the type `IntObj`. The final line declares the set `IntValues` to contain primitive integer values. Many objects in the computation may have the same consistency properties. For example, all of the nodes in a linked list usually have the same consistency properties. We expect that developers will typically use sets to group together all objects that have the same consistency properties.

We expect that developers will use relations to model the values of fields and referencing relationships. For example, a developer might use a set to store the nodes in a linked list and a relation to model the fields that link the nodes together.

2.2 Translating Data Structures into Relations

The first step in the repair process translates the concrete data structures into the sets of objects and relations in the relational model. Sets contain either objects or primitive values depending on the declared type of the set. Conceptually, there are two different types of relations in the model: relations that model object fields that store primitive values, and relations that model the referencing relationships between objects. Relations that model array indexes are an interesting special case; while the underlying field stores a primitive value, the relation may relate the location of the index to the indexed element in the array.

Our approach to translating data structures into relations has two major components: decoding the bits in the heap into objects and tracing out the referencing relationships between these objects to construct the sets and relations in the relational model.

2.2.1 Decoding the Heap

Applications represent their data structures as a string of bits in memory. The first step of the repair process decodes this string of bits into objects with fields. The repair system could use C structure declarations for this step, but this approach has some limitations. Instead, we have developed a structure definition language based on C structure declarations with extensions added to address the limitations. The first limitation is that C structure definitions do not support dynamically sized arrays. We have developed an extension that allows the sizes of these arrays to be computed from integer values stored in the decoded fields of data structures. The second limitation is that C structure declarations do not support packed arrays of bits. We have developed an extension to support packed arrays of bits. Finally, C structure declarations do not support inheritance. We have developed extensions to support two types of inheritance: structural inheritance and extensional inheritance. Structural inheritance allows a developer to define types that leave a region in the data structure undefined, and then incrementally refine part of that region in that type in a second type definition that declares fields in these undefined parts. Extensional inheritance allows the developer to define a type that adds fields to a previous type definition.

```
structure Obj {
}
structure IntObj subclass of Obj {
   int value;
}
IntObj *ptr;
```

Figure 2-3: Example Structure Definitions

Figure 2-3 presents the structure definitions for the example. The first structure definition declares the type `Obj` to contain no fields. The second structure definition declares that the type `IntObj` extensionally inherits from the type `Obj` and adds the primitive integer field `value`. The last line of this figure declares that the application

has a variable `ptr` that references a `IntObj` object.

2.2.2 Translating the Decoded Heap into a Relational Model

After decoding the bits in the heap, the repair algorithm finds and classifies objects and constructs relations from the information stored in the fields of objects. The basic strategy is to construct the sets and relations by starting from a set of roots to trace out the pointers in the heap. The repair algorithm classifies objects into sets based on both the path taken through the heap to reach the object and the contents of the objects.

Our repair algorithms use a set of developer-provided model definition rules to trace out the referencing relationships in data structures to find and classify objects.

Each model definition rule has the following components in order:

- Optional quantifiers over sets and/or relations
- A guard that tests a condition on the concrete data structure and/or the abstract model
- A right arrow that separates the guard from the inclusion condition
- An inclusion condition that specifies an object (or tuple) to add to a set (or relation) if the guard is true.

To trace out the pointers in a recursive data structure, a developer would typically use a pair of model definition rules: one rule handles the base case and the other handles the recursive case. The base case model definition rule adds the root of the data structure to a set. The recursive case model definition rule then quantifies over this set, checks if a developer-specified pointer in the quantified object is non-null, and if so adds the pointer's value to the set. We designed the model definition rules to be both easy to analyze for common usage patterns yet powerful enough to support sophisticated translations.

Figure 2-4 presents the model definition rules for the example. The first model definition rule checks to see if `ptr` is non-null, and if so, adds the object referenced

```
1. ptr != null => ptr in AllInts
2. (ptr != null) and (ptr.value >= 0) => ptr in NonNegInts
3. for n in AllInts, true => n.value in IntValues
```

Figure 2-4: Example Model Definition Rules

by `ptr` to the `AllInts` set. The second model definition rule checks to see if `ptr` is non-null and if `ptr.value` is greater than or equal to zero, and if so, adds the object referenced by `ptr` to the `NonNegInts` set. Notice that the first two model definition rules do not have quantifiers as they operate directly on a pointer in the root set. The final model definition rule quantifies over all objects in the `AllInts` set and adds the contents of the `value` field of these objects to the `IntValues` set. Since the third model definition rule does not need to test any conditions, we have used `true` as the guard.

Ensuring Termination of Model Construction Sometimes it is useful to construct a set of objects that do not satisfy some property. For example, a developer may define an active set of objects that participate in a given data structure, and then use negation to specify a free set that contains objects that are not in the active set. Negation complicates model construction because it may introduce a non-monotonicity into the fixed point computation that constructs the relational model. Consider the model definition rules `!o in S => o in S'` and `true=>o in S`. The issue with negation is that the first model definition rule may initially place the object referenced by `o` in `S'` because `o` is not in `S`. If a second model definition rule then adds `o` to `S`, this causes the guard of the first model definition rule to become false even though the this rule has already added `o` to `S'`. Note that if the model construction algorithm had evaluated the rules in the opposite order, the algorithm would only add `o` to `S`. Notice that there are two solutions. The model construction algorithm generated the second solution because it evaluated a negated inclusion constraint on a partially constructed and therefore added `o` to `S'`. The problem is that it is possible that only one of these models satisfies the consistency constraints, and

therefore whether a data structure is consistent may depend on the order in which the model definition rules is evaluated. Our algorithm eliminates the other solutions by ordering the evaluation of the model definition rules to ensure that the model construction algorithm completely constructs all of a model definition rule's negated set and relation dependences before evaluating the model definition rule. In the example, this ordering would ensure that the second model definition rule was evaluated before the first model definition rule.

Consider the model definition rule `!o in S => o in S`. This model definition rule has a negated dependence on the set it constructs, and therefore the model construction algorithm cannot completely construct `S` before evaluating the rule. To address this issue, we restrict negation to be used only when a model definition rule's negated predicate does not depend directly on the set or relation constructed by that model definition rule or indirectly on this set or relation through the actions of other model definition rules. In the current example, this restriction disallows the model definition rule `!o in S => o in S`. In general, this restriction allows the model construction algorithm to completely construct all of a model definition rule's negated set and relation dependences before evaluating the model definition rule.

Our repair algorithm also supports the creation of sets or relations that contain primitive values. The primary issue with supporting primitive values is the possibility of constructing an infinite model. For example, a model definition rule could state that the successor of every integer in a set is also in the set. To address this issue, our repair algorithm places limitations on the synthesis of new primitive values in the inclusion condition of model definition rules that quantify over a set or relation that may grow larger as a result of the action of this model definition rule. These restrictions serve to ensure that the generated model is finite.

Ensuring Basic Representation Consistency The model construction process ensures the basic representation consistency of the data structures as follows. It ensures that data structure references point to allocated memory. If it discovers a pointer to unallocated memory, it writes a zero value to that pointer. It also ensures

that objects do not illegally overlap with other objects. If the repair algorithm finds illegally overlapping objects, it writes a zero value over the reference to one of these objects. Finally, the developer can write model definition rule guards that ensure that the model construction step does not place illegal values in sets or relations.

2.3 Specifying Consistency Constraints

Once the repair algorithm constructs the abstract model, it needs to know what constraints the model should satisfy. The developer expresses consistency constraints in terms of the sets of objects and relations in the model.

```
1. size(AllInts)=1
2. size(NonNegInts)=0
```

Figure 2-5: Example Consistency Constraint

Figure 2-5 presents the consistency constraint for the example. This example contains two consistency constraints. The first set size constraint `1. size(AllInts)=1` ensures that the `AllInts` set contains exactly one object. The effect of this constraint is to ensure that the `ptr` variable always points to an `IntObj` object. The second set size constraint `2. size(NonNegInts)=0` ensures that the `NonNegInts` set is empty. The effect of this constraint is to ensure that the `ptr` variable only points to an `IntObj` object that stores a negative integer value.

2.3.1 Classes of Basic Propositions

Our repair system supports boolean combinations of the following classes of basic propositions:

- **Numerical Inequalities:** Developers often use relations to model the values of a data structure field or variable. Numerical inequality basic propositions allow the developer to express numerical inequalities over the primitive field values that these relations model. An example numerical inequality is `x.R>1`.

- **Pointer Equalities:** These basic propositions express pointer equalities. Developers typically use these constraints to ensure local topological constraints, such as consistency between forward and backward pointers. An example pointer inequality is `n.Next.Prev=n`. The `Next` and `Prev` relations in this example are used to model the `next` and `prev` fields in a double linked list.

- **Set Size Constraints:** These basic propositions express constraints on the sizes of sets. In general, developers use these constraints to ensure that data structures exist. The previous example contains the constraint `size(AllInts)=1`, which ensures that the `AllInts` set contains exactly one `IntObj` object. This constraint ensures that the `IntObj` object pointed to by the `ptr` variable exists.

- **Topology Constraints:** These basic propositions express constraints on how many objects a relation (or its inverse) maps a given object to or from. Developers typically use these constraints to express local topological properties of the heap — specifically what the in-degree or out-degree of a given object is. For example, the topology constraint `size(n.~Next)<=1` ensures that a node in a linked list is referenced at most once by the next field of another node in the linked list.

- **Inclusion Constraints:** These basic propositions express constraints on whether an object (or tuple) is in a set (or relation). An example of an inclusion constraint is `n in Nodes`. This constraint ensures that `n` is a member of the set `Nodes`. These propositions are useful for relating an object's membership in a set to some other condition (perhaps whether a flag is set in the object).

We chose this set of basic propositions because we could express many interesting data structure properties using them and it was easy to generate repair actions for the individual basic propositions.

2.3.2 Quantifiers

Our consistency constraint language supports guarded universal quantifiers over sets and relations. Quantifiers allow developers to write consistency constraints that apply to all objects in a set or to all tuples in a relation. We have restricted the form of the quantifiers to simplify the repair algorithm. In particular, we only allow guarded universal quantifiers to appear to the left of the body of the constraint, and we disallow existential quantifiers. These restrictions simplify the evaluation and the repair of the constraint, as the repair algorithm does not need to perform a search to find the appropriate quantifier binding. One drawback of this restriction is that disallowing existential quantifiers complicates the expression of properties about incoming reference to an object. For example, the formula `for n in Nodes, n in Root or exists parent in Tree, parent.left=n or parent.right=n` specifies that all nodes (except the root node) in the tree have a parent. We address this issue by supporting the inverse operation on relations. The developer can use relation inverses to express constraints about the existence of an incoming reference to an object. For example, we can rewrite the previous constraint using relation inverses as `for n in Nodes, n in Root or size(n. Left)>=1 or size(n. Right)>=1`.

The existential quantifier can also specify that at least one element in a set satisfies a given property. For example, the constraint `exists tail in List, size(tail.Next)=0` ensures that a list has a tail node that doesn't reference any other nodes. While this type of property cannot be specified in our model construction language, the developer can write a model definition rule to place the tail of a linked list in its own set, and the developer can specify that this set contains one element.

2.4 Repairing the Model

Consider the first constraint `size(AllInts)=1` from the example. If the variable `ptr` was null, the model construction step would construct an empty `AllInts` set, and the consistency checking step would detect a violation of the constraint `size(AllInts)=1`. The repair algorithm would generate a model repair that creates a `IntObj` object and

adds this object to the `AllInts` set.

The goal is to generate a sequence of operations that add or remove objects (or tuples) to or from sets (or relations). Our repair algorithm constructs this sequence of repair actions as follows. It repairs a constraint by choosing a conjunction in the disjunctive normal form (DNF) of a constraint and performs individual repair actions to the basic propositions in the conjunction making that conjunction true.[1] Note that making any conjunction in the DNF form of a constraint true makes the entire constraint true. Our repair algorithm locally selects a conjunction to make true, using a cost function to guide the selection of the conjunction. This function assigns costs for making each of the basic propositions true. While this cost mechanism was originally designed to guide the operation of the repair algorithm to select repairs that minimally change the data structure, we have extended it to allow developers to specify preferences for certain repair actions. When repairing a constraint, the repair evaluates the total cost of the repair actions that each conjunction choice will require and then chooses the conjunction with the smallest cost to repair. This greedy algorithm enables us to avoid search during repairs, but comes at the cost of potentially performing lower quality repairs. This could be alleviated by combining our current technique with a bounded search to select repair actions.

One limitation of this repair strategy is that it only considers a single basic proposition when generating a repair action. We chose this design to simplify the repair algorithm. However, this simplicity comes at a cost: our repair algorithm is unable to generate repair actions that satisfy multiple constraints at once. One example of such a specification is a system of linear equations. A successful repair strategy for linear equations needs to coordinate updates to multiple equations when generating repair actions. Our system would only generate repair actions for individual equations; such actions would violate other equations in such a specification. As a consequence, our specification compiler would fail to generate a repair algorithm for such a specification,

[1]The disjunctive normal form of a constraint is a normal form where the constraint is expressed as a disjunction (or operation) of conjunctions (and operation) of the basic properties in the constraint language.

since our repair algorithm generation process would only generate repair algorithms that may fail to terminate in some cases.

2.5 Translating Abstract Repairs into Data Structure Updates

The final component of the repair process translates the abstract repairs into updates to the concrete data structures. For example, to translate a model repair that adds an object to a set, the repair algorithm finds the model definition rules that construct the given set, and selects one of these model definition rules to add the object to the set. The goal at this point is to make the guard of the model definition rule true in such a way that the model definition rule adds the object to the set. Guards of model definition rules are composed of boolean combinations of predicates that test either the concrete data structure or the abstract model. The repair algorithm uses the same technique to satisfy the guard as it used to satisfy a model consistency constraint except that it manipulates the concrete data structure in this case - specifically, it converts the guard in the constraint to disjunctive normal form, selects a conjunction to satisfy, and then performs a series of updates to the concrete data structure to satisfy that conjunction. In general, our algorithm finds the model definition rules whose inclusion condition (right hand side) constructs the relevant set or relation, and then manipulates the guard (left hand side) of the model definition rule to achieve the desired action. Note that the inclusion condition of a model definition rule can have only one of two forms: either it specifies an object to add to a set or it specifies a tuple to add to a relation.

In some cases, the expressions in the inclusion condition of the model definition rule may not evaluate to the object or tuple that the model repair adds to the set or relation. In this case, the repair algorithm may also manipulate the fields in the expressions that appear in the inclusion condition of the model definition rule to make the expression equal to the object (or tuple) that the model repairs adds to the set

(or relation). Specifically, it sets the last field in the expression equal to the object or primitive value to be added to the set or relation. Note that if the last field is an array and the object (or half of the tuple) to be added to the set (or relation) is also from the same array, then the repair algorithm sets the index into the array to the equal to the index of the object to be added. Finally, it is possible that the sets that the model definition rule quantifies over do not contain the object that the corresponding quantified variable is bound to. In this case, the repair algorithm may also need to add or remove objects (or tuples) to or from the sets (or relations) that appear in the quantifiers of the model definition rule.

To remove an object from a set, the repair algorithm generates updates that falsify the guards of all the model definition rules that may have added the object to the set. The repair algorithm may also remove objects (or tuples) from the sets (or relations) that the model definition rule quantifies over to prevent that model definition rule from adding an object to a set. If the repair algorithm needs to add or remove tuples, it uses a similar strategy.

2.5.1 How Additional Changes to the Model Occur

It is unfortunately possible that a given data structure update may change the model in additional ways that go beyond the desired change. We identify two mechanisms for these additional changes: changes that the data structure update performs to the data structures can cause model definition rules to add or remove objects (or tuples) to or from sets (or relations) and changes that the data structure update causes to the abstract model can cause model definition rules to add or remove objects (or tuples) to or from sets (or relations). We next examine the example to understand why this may happen. Consider a model repair that adds a new `IntObj` object to the `AllInts` set. The repair algorithm would translate this model repair into a concrete data structure update that sets the variable `ptr` to point to the newly allocated `IntObj` object. If the `value` field of the newly allocated `IntObj` object happens to be non-negative, this update would also satisfy the guard of the second model definition rule shown in Figure 2-4. Therefore, the data structure update may cause the newly

allocated `IntObj` object to be placed in the `NonNegInts` set. This is an example of a general class of cascading changes caused by changing state that is constrained by other model definition rules. In general, updates to concrete data structures may affect any model definition rules that read the updated state.

This update also causes the third model definition rule to add a primitive integer value to the `IntValues` set. This occurs because the data structure update causes a new object to be added to the `AllInts` set and the third model definition rule then quantifies over this set. This is an example of a more general class of cascading change caused by changes to the relational model. In general, changes to the relational model may cause any model definition rule that quantifies over the changed set (or relation) or tests membership in the changed set (or relation) to be satisfied or falsified.

These unintended cascading changes can cause additional constraints to be violated. After the repair algorithm performs a concrete data structure update, our repair algorithm rebuilds the model, and rechecks the constraints. If the repair algorithm finds further violations, it repairs these violations. It repeats this process until all violations have been repaired.

2.6 Termination of the Repair Algorithm

One issue with this repair strategy as outlined above is that it can fail to terminate. For example, a repair algorithm could repair a violation of one constraint and in the process violate a second constraint. The repair algorithm could then repair the violation of the second constraint and in the process violate the original constraint, leading to an infinite repair loop.

The specification compiler is the part of our system that compiles the data structure consistency specifications into C code that implements the repair algorithm. Our specification compiler reasons as follows to ensure that all repairs terminate.

2.6.1 Repair Dependence Graph

We have developed the *repair dependence graph* to reason about termination of the repair process. The repair dependence graph captures dependences between repair actions and constraints. Note that events in an individual repair follow paths in the graph. For example, suppose that the repair algorithm decides to repair a conjunction. It then would perform model repairs followed by data structure updates to implement the model repairs. Note that the data structure updates may change the scopes of the model definition rules, and as a result the repair algorithm may perform compensation updates to counteract these scope changes. All of these chains of events correspond to the paths in the repair dependence graph that start from the model conjunction.

Therefore, the specification compiler can determine what constraints a given repair action may (transitively) violate by tracing out the edges in this graph. By examining what part of the graph is reachable from a given repair action, the specification compiler can determine all possible cascading constraint violations and repair actions that a given repair action could transitively cause. The absence of cycles in this graph implies that there are no infinite repair loops, and therefore that the repair process eventually terminates. We give a correctness argument of this implication in Section 6.5.

The basic idea is that the repair dependence graph has a node for each conjunction of each constraint and a node for each repair action. The graph contains directed edges to keep track of dependences between the repair actions and the constraints. The specification compiler places an edge from a conjunction node to a repair action node if satisfying the conjunction corresponding to the conjunction node may require the repair algorithm to perform the repair action corresponding to the repair action node. The specification compiler places an edge from a repair action node to a conjunction node if performing the repair action corresponding to the repair action node may falsify the conjunction corresponding to the conjunction node.

Figure 2-6 presents the repair dependence graph for this example. The elliptical nodes correspond to conjunctions in the disjunctive normal form of a constraint. The

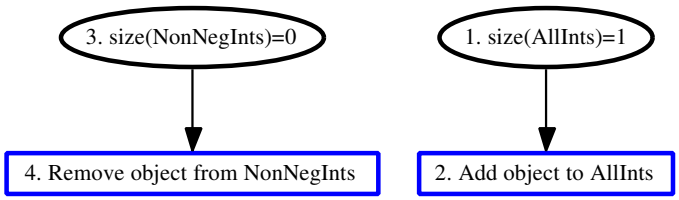


Figure 2-6: Repair Dependence Graph

elliptical node labeled `1. size(AllInts)=1` corresponds to the first constraint. The elliptical node labeled `3. size(NonNegInts)` correspond to the second constraint. The rectangles correspond to repair actions. The rectangular node labeled `2. Add object to AllInts` corresponds to a model repair that adds an object to the `AllInts` set to satisfy the first consistency constraint. The rectangular node labeled `4. Remove object from NonNegInts` corresponds to a model repair that removes an object from the `NonNegInts` set to satisfy the second consistency constraint.

The edges in this graph capture dependences. The edges from elliptical conjunction nodes to rectangular repair action nodes capture the dependence that repairing violations of the corresponding conjunctions requires performing the corresponding repair actions. The edges from the rectangular repair action nodes to the elliptical conjunction nodes captures the dependence that performing the corresponding repair action may violate the corresponding conjunctions.

2.6.2 Cascading Changes

One complication with this approach is that data structure updates can cause additional changes to the model as described in Section 2.5.1. This can happen because other model definition rules may depend on the state modified by a data structure update, and therefore, the data structure update may cause such model definition rules to add or remove an object (or tuple) to or from a set (or relation). This can also happen if model definition rules quantify over or test membership in the set (or relation) that the data structure update changes, and therefore, the data structure update can cause additional additions or removals of objects (or tuples) to or from sets

(or relations). These changes can propagate even further through the same cascading mechanism.

The solution to this problem is to fix the repair dependence graph so that it tracks these cascading changes. Our solution adds a pair of nodes for each model definition rule. One node represents a change that causes the model definition rule to add an object (or tuple) to a set (or relation), while the other represents a change that causes the model definition rule to remove an object (or tuple) from a set (or relation). We refer to these nodes as scope increase and scope decrease nodes, respectively.

We also add nodes to represent data structure updates. These are necessary because there are often multiple ways to translate a model repair into a data structure update, and each of these data structure updates may potentially cause different additional changes to the model. The specification compiler uses the data structure update nodes to keep track of the different effects of each of the data structure updates that implement a given model repair.

We add the following edges to capture the dependences between model repair actions, data structure updates, model definition rules, and constraints:

- **Edges from Model Repair Nodes:**

  To translate a model repair to the concrete data structure, the repair algorithm performs data structure updates on the concrete data structure. The specification compiler tracks this dependence by placing an edge from a model repair node to the data structure update nodes that implement the given model repair.

  The changes that a model repair makes to the relational model may cause model definition rules to add or remove additional objects (or tuples) from sets (or relations). The specification compiler tracks these dependences by placing an edge from a model repair node to a scope increase or decrease node if the corresponding model repair may cause the corresponding model definition rule to add or remove, respectively, an object (or tuple) to or from a set (or relation).

- **Edges from Data Structure Update Nodes:**

A data structure update may change the state that other model definition rules depend on, causing them to add or remove objects (or tuples) to or from sets (or relations). The specification compiler tracks these dependences by placing an edge from a data structure update node to scope increase or decrease nodes if the corresponding data structure update may cause the corresponding model definition rule to add or remove, respectively, an object (or tuple) to or from a set (or relation).

- **Edges from Scope Nodes:**

  Changes in the scope of one model definition rule may result in further changes in the scopes of other model definition rules. The specification compiler tracks these dependences by placing an edge from one scope node to a second scope node if a change in the scope of the model definition rule corresponding to the first scope node may result in a change in the scope of the model definition rule corresponding to the second scope node.

  Finally, changes in the scope of a model definition rule cause changes to the model that may falsify constraints. The specification compiler tracks these dependences by placing an edge from a scope increase or decrease node to a conjunction node if increases or decreases, respectively, of the corresponding model definition rule may cause the conjunction to be falsified.

Recall that Figure 2-4 presents the model definition rules for the example. Our tool analyzes these model definition rules to translate the model repairs into data structure updates. Recall that the repair algorithm translated a model repair that added the newly allocated object to the `AllInts` set into a concrete data structure update that sets the variable `ptr` to point to the newly allocated `IntObj` object, and that this data structure update could potentially add the object to the `NonNegInts` set.

Figure 2-7 presents the repair dependence graph with nodes added to model data structure updates and model definition rule scope changes. The dashed rectangles are data structure update nodes, and the dashed ellipses are scope increase nodes. The

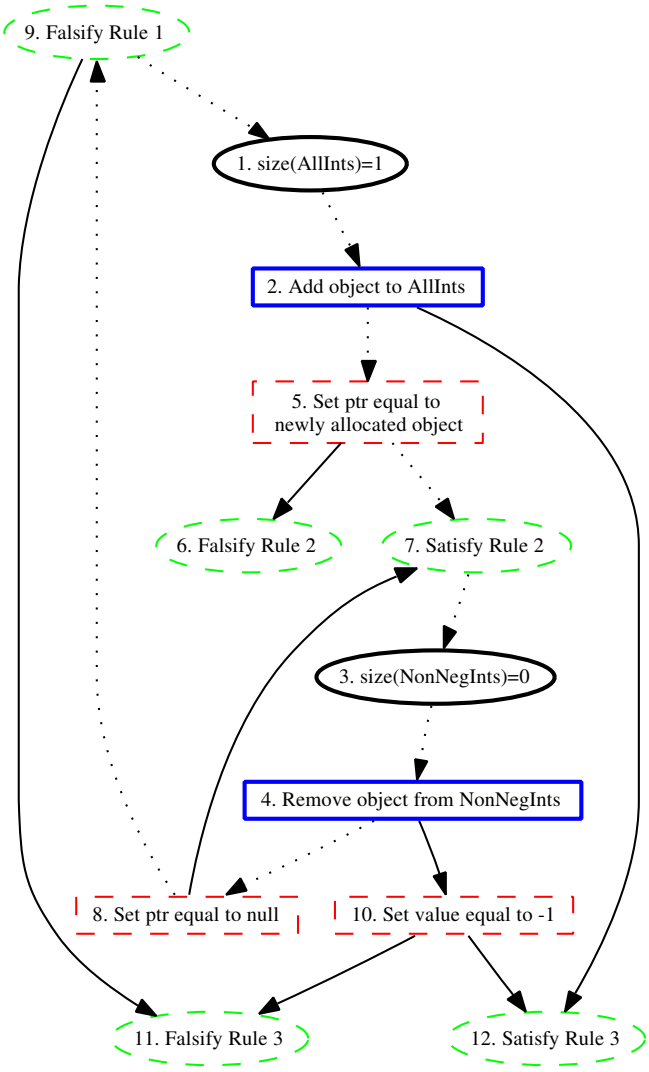


Figure 2-7: Repair Dependence Graph with Updates

graph construction algorithm generates the scope increase node `7. Satisfy Rule 2` because setting the variable `ptr` to the newly allocated object may cause the second model definition rule to add the object to the `NonNegInts` set if its `value` field is non-negative. Note that the edge from the data structure update node `8. Set ptr equal to null` to the scope increase node `7. Satisfy Rule 2` is present because our graph construction algorithm is conservative. The analysis places this edge because setting the variable `ptr` to null may affect the proposition `ptr.value!=null` in the second model definition rule. Our graph construction algorithm places edges from this scope increase node to the conjunction node `3. size(NonNegInts)=0` be-

cause adding an object to the `NonNegInts` set will violate the consistency constraint `size(NonNegInts)=0`. The algorithm places edges from the rectangular model repair nodes to the dashed rectangles data structure update nodes, because the repair algorithm performs these data structure updates to implement the model repairs.

2.6.3 Pruning Nodes

As we discussed earlier, the absence of cycles in the repair dependence graph implies that the repair algorithm will terminate. However, the graph in Figure 2-7 contains the cycle indicated by the dotted edges. Note that the specification compiler has translated the model repair represented by the model repair node `4. Remove object from NonNegInts` into two different data structure updates corresponding to the data structure update nodes `8. Set ptr equal to null` and `10. Set value equal to -1`. The generated repair algorithm only needs to perform one of the two data structure updates to implement the repair. Therefore, we can prune the data structure update node `8. Set ptr equal to null` from the graph because the repair algorithm never needs to perform this update. We present the resulting acyclic graph in Figure 2-8. Since this graph is acyclic, the corresponding repair algorithm terminates.

2.6.4 Compensation Updates

Our translation of model repairs into data structure updates is based on analyzing a single model definition rule to synthesize a data structure update. One downside of this approach is that it ignores the effect of this update on other model definition rules. We could potentially synthesize more precise updates by simultaneously considering multiple model definition rules.

We created *compensation updates* to partially address this issue. Compensation updates prevent undesired increases in the scope of a model definition rule by falsifying the guard of the model definition rule or by removing an object (or tuple) from one of the sets (or relations) that appear in the model definition rule's quantifiers. The repair algorithm implements a model repair by using a data structure update and

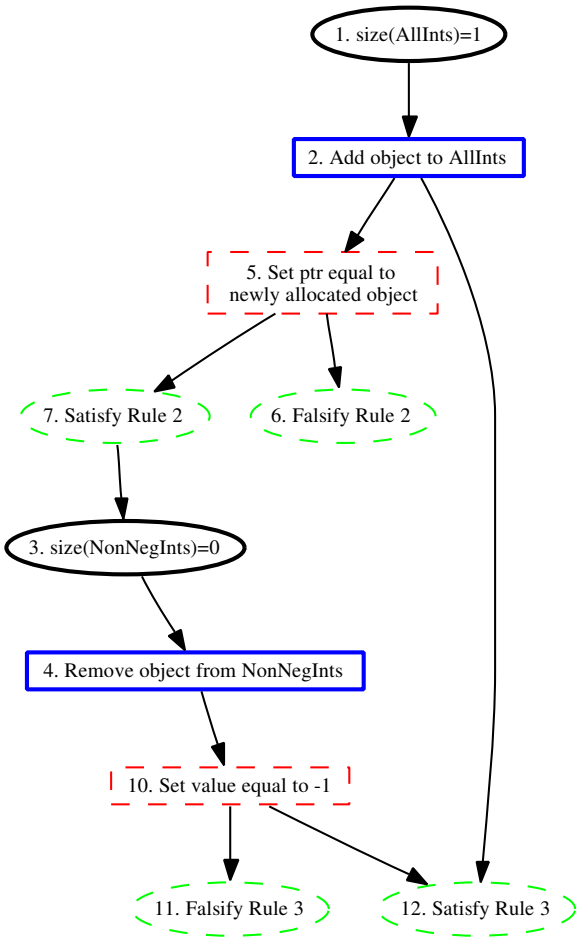


Figure 2-8: Pruned Repair Dependence Graph with Updates

a set of compensation updates. To support compensation updates we have added further nodes and edges to the repair dependence graph as described in Section 6.

2.7 Scope

While we successfully used data structure repair to enable several applications to recover from otherwise fatal data structure inconsistencies, the approach has the following limitations:

- **Parsing of Data Structures:** The current system is designed to repair data structures that are stored in the fields of objects. Some systems store data as a long textual string that must be parsed. Examples of such data structures

include XML and HTML documents. These data structures require a more advanced approach to parsing: a parser must recognize textual tokens in the document and then combine these tokens into expressions. Other data structures such as the Windows NT file system store data as a sequence of variable length records. To decode these data structures, the system must parse the variable length records. Our model building approach is currently unable to parse such data structures.

- **Support for Primitive Types:** The current system does not support some of the commonly used primitive types. For example, it does not support 64 bit integers, unsigned integers, or floating point numbers. As a result, many invariants cannot be expressed in our system. However, it should be straightforward to add support for additional primitive types into our system.

- **Repair Generation:** Even if the developer is able to write a given consistency constraint in the specification language, the specification compiler may not be able to generate a repair algorithm for the given specification. This can occur because the specification compiler is unable to generate the necessary repair actions. For example, the current repair system only generates repair actions that repair a single constraint violation. Repairing some sets of constraints may require repair actions that simultaneously solve multiple constraints.

- **Termination Analysis:** Even if the specification compiler can generate the necessary repair actions, it is possible for the specification compiler to be unable to verify that the generated repair algorithm terminates. This can occur because the conservative repair dependence graph construction algorithm places extraneous edges in the graph, or because repair dependence graph-based approach is insufficient to show that a given repair algorithm terminates. This can happen if the repair dependence graph contains cycles that the repair algorithm only repairs a bounded number of times.

- **Developing Specifications:** Developing specifications may prove to be a significant burden in some cases. For example, the effort required to develop a specification for a large legacy application may preclude using our data structure repair tool. While we have written some specifications for large applications, these specifications were partial and they described easily understood core data structures. Complete consistency specifications for complicated, poorly documented data structures may be very difficult to write. Most of the difficulty in such an effort would likely arise in understanding how the application manipulates data structures. In particular, the developer must understand what consistency properties should hold for the data structures in the application, and when these consistency properties should hold. Furthermore, the developer must annotate the application to indicate where a given specification should hold.

- **Runtime Overhead:** The repair algorithm uses dynamic consistency checking to detect data structure inconsistencies. Every time it checks the consistency properties, it traverses the data structures, builds an abstract model of the data structure, and then checks the consistency properties on this abstract model. For applications that manipulate data structures with many consistency properties that must be checked or that require very frequent consistency checking, the checking overhead can be significant. It may be possible to reduce this checking overhead by a combination of static analysis and utilizing hardware page protection mechanisms to incrementally check consistency constraints.

- **Types of Errors:** Data structure repair only addresses data structure corruption errors. Other types of software errors are out of the scope of this technique. Techniques such as recursive restartability have been developed to handle other types of errors [9]. Recursive restartability can handle errors that corrupt the transient state or that cause the application to follow the wrong execution path by rebooting the affected component. However these techniques cannot handle corruption of persistent data structures or recurrent problematic inputs. Hybrid

recovery strategies that incorporate repair should be able to effectively recover from a larger class of errors.

- **Applicability:** While data structure repair may prevent an application from crashing, it is still possible for the application to produce unacceptable results. For example, incorporating repair into cryptography routines is likely to result in either insecure and/or unreadable messages. Further experience using data structure repair may help us develop a better intuition of what applications are appropriate for data structure repair.

2.8 Summary

Our approach relies on a developer-provided data structure consistency specification. Our repair algorithm uses this specification to decode the string of bits in the heap into objects with fields, to construct the sets and relations in an abstract relational model of the data structures, then to check all of the consistency constraints on this model. If the repair algorithm finds a violated constraint, it performs repairs to this model, and then translates these model repairs into data structure updates that modify the bits in the heap. It repeats this process until all of the constraints are satisfied. Our system statically analyzes the generated repair algorithm to verify that it terminates.

Chapter 3

File System Example

Having described at a high level how our repair algorithm works, we now present a more extensive example, which implements a simple file system. File systems are a useful example to consider because they have many rich consistency properties that must be satisfied to ensure proper operation; they are prone to inconsistencies due to hardware faults, user errors, and software errors; they are persistent data structures and therefore inconsistencies persist across reboots; and finally, it is typically unacceptable to restore a file system's consistency by reinitializing it. Repair is particularly useful for file systems because an inconsistency in a file system can cause valuable data to become unreadable. For these reasons, file system developers have traditionally provided hand-coded utilities such as chkdsk [3], fsck [5], or scandisk to detect and repair inconsistencies. This example shows how we automatically generate a repair algorithm for a file system from a specification.

Figure 3-1 presents a graphical representation of the file system in our example. The file system consists of an array of disk blocks. In order to quickly allocate new blocks, the file system keeps a table of which blocks are in use. The file system reserves a block, the bitmap block, for storing this table. The file system reserves another block to store the inode table, which keeps track of which blocks store the information for a particular directory or a file. Finally, the first block in the file system is reserved for the superblock, which is a special block that stores critical file system configuration information. For example, it stores the size of the blocks in

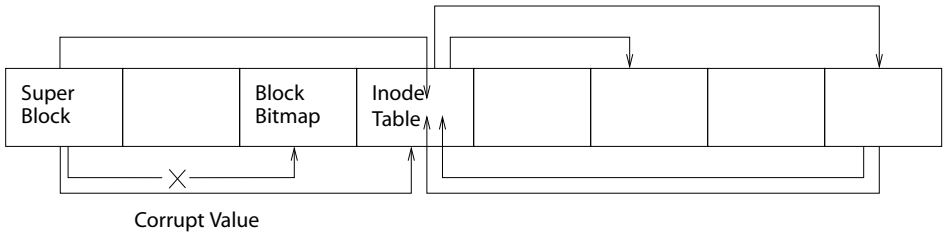


Figure 3-1: Inconsistent File System

the file system, which block contains the block bitmap, and which block contains the inode table.

The particular file system shown in Figure 3-1 has an inconsistency. As indicated by the 'X', the bitmap block location stored in the superblock has been corrupted and no longer stores the correct location of the bitmap block. As a result, future block allocations will fail.

3.1 Consistency Specification

The data structure consistency specification consists of two parts: a part that specifies a translation from the concrete data structures into an abstract model, and a part that specifies consistency constraints that this abstract model must satisfy. The translation part of the specification consists of the data structure declarations in Figure 3-2 (these declarations specify the physical layout of the data structures that comprise the file system), the set and relation definitions in Figure 3-3 (these definitions specify the sets and relations in the model of the file system), and the model definition rules in Figure 3-4 (these rules specify how to construct the abstract model from the data structures).

3.1.1 Structure Declaration

The first part of the translation specification specifies how the data structures are physically laid out in memory. This part of the specification uses a structure definition

```
structure Disk {
     Block  b[d.s.numberofblocks];
     label b[0]: SuperBlock s;
}
structure Block {
     reserved byte[d.s.blocksize];
}
structure SuperBlock subtype of Block {
     int numberofblocks;
     int numberofinodes;
     int blocksize;
     int rootdirectoryinode;
     int blockbitmap;
     int inodetable;
}
structure BlockBitmap subtype of Block {
     bit bitmap[d.s.numberofblocks];
}
structure DirectoryBlock subtype of Block {
     DirectoryEntry de[d.s.blocksize/128];
}
structure DirectoryEntry {
     byte name[124];
     int inode;
}
structure InodeTable subtype of Block {
     Inode itable[d.s.numberofinodes];
}
structure Inode {
     int block[12];
     int referencecount;
}

Disk *d;
```

Figure 3-2: Structure Definitions

language that is similar to C structure definitions with a few extensions. One of these extensions allows the developer to specify variable length arrays in which the length is stored in a data structure. We now examine the data structure definitions in Figure 3-2 in more detail.

The file system consists of an array of `Block` objects. The first line in the Ďisk structure definition declares that the `Disk` object contains this array of blocks. The second line declares a label `s` for the first block in the file system and that this block has the type `Superblock`. Note also that the size of the array of blocks is given by the expression `d.s.numberofblocks`. The `d` variable in this expression refers to the `Disk *d;` declaration at the bottom of the figure. This declaration declares that the variable `d` points to a `Disk` object and that this value is provided to the repair algorithm from the underlying application. The `s` refers to the label `s` in the `Disk` type declaration, and the `numberofblocks` refers to the `numberofblocks` field in the `SuperBlock` type declaration.

The `Block` structure definition specifies that a block is `d.s.blocksize` bytes long. The `reserved` keyword indicates that the Block structure doesn't define how this space is used. This value is stored in the `SuperBlock` of the file system.

The `SuperBlock` stores the basic layout parameters for the file system: it stores the number of blocks and inodes, the size of the blocks, the inode that contains the root directory, and the locations of the block bitmap and the inode table. Note that the first line of the `Superblock` declaration `structure SuperBlock subtype of Block` declares that the `SuperBlock` type structurally inherits from the `Block` type. This indicates that the fields declared in the ŠuperBlock declaration can refine the `reserved` space in the `Block` declaration, and that objects of the `SuperBlock` type have the same size as objects of the `Block` type.

The `BlockBitmap` contains an array of bits: one bit for each block in the file system. If the block is used, the corresponding bit is set to `true`. Otherwise, if the block is free, the corresponding bit is set to `false`. This array of bits enables the file system to efficiently allocate unused blocks.

The `DirectoryBlock` contains an array of directory entries. Each `DirectoryEntry`

contains the name of a file and a reference to the file's inode.

The `InodeTable` contains the array of inodes in the file system. Each `Inode` stores references to the blocks that contain the file's data and a reference count. This reference count stores a count of how many directory entries reference the inode.

3.1.2 Model Definition

The next part of the translation specification specifies how to construct the sets and relations in the model from the objects discussed in the previous section. This part of the specification first declares the sets and relations in the abstract model and then specifies how to construct the sets and relations in the abstract model from the concrete data structure. The model construction phase places objects in the data structure into the appropriate sets and constructs the relations.

Figure 3-3 presents the set and relation declarations for our abstract model of the file system example. The declaration `set Block of Block : UsedBlock | FreeBlock` declares that the set `Block` contains data structures of type `Block` and contains two subsets: `UsedBlock` and `FreeBlock`. In general, the set declarations in Figure 3-3 are of the form `set` S `of` T : $S_1|\ldots|S_n$. Such a declaration specifies that the set S in the model contains objects of type T (these types are either base types such as `int` or structures) and that the sets $S_1, \ldots, S_n$ are subsets of the set S. The declaration `relation InodeOf: DirectoryEntry -> UsedInode` declares that the relation `InodeOf` relates objects in the set `DirectoryEntry` to objects in the set `UsedInode`. In general, the relation declarations in Figure 3-3 are of the form `relation` R : S_1 `->` S_2. Such a declaration specifies that the relation R relates the objects in set S_1 to the objects in set S_2.

Conceptually, the model definition rules in Figure 3-4 specify how to traverse the data structures to build the sets and relations in the abstract model. The model definition rules are of the form `Quantifiers, Guard => Inclusion Condition`. Each rule specifies quantifiers that identify the scope of the variables in the body. The inclusion condition specifies an object (or tuple) that must be in a specific set (or relation) if the guard is true. The repair algorithm evaluates the model definition rules

```
set Block of Block : UsedBlock | FreeBlock
set UsedBlock of Block : SuperBlock | FileDirectoryBlock | InodeTable |
  BlockBitmap
set FileDirectoryBlock of Block :  DirectoryBlock | FileBlock
set UsedInode of Inode : FileInode | DirectoryInode
set DirectoryInode of Inode : RootDirectoryInode
set DirectoryEntry of DirectoryEntry
relation InodeOf: DirectoryEntry -> UsedInode
relation Contents: UsedInode -> FileDirectoryBlock
relation BlockStatus: Block -> int
relation ReferenceCount: UsedInode -> int
```

Figure 3-3: Set and Relation Declarations

on the concrete data structure to generate the abstract model. Figure 3-4 presents the model definition rules for the file system example.

The first model definition rule places the first block in the file system in the `SuperBlock` set. The next two model definition rules place the `d.s.blockbitmap` and `d.s.inodetable` elements of the block array into the `BlockBitmap` and `InodeTable` sets, respectively. These three model definition rules identify key blocks in the file system and place them into sets.

The `rootdirectoryinode` field of the `SuperBlock` stores the index of the root directory in the inode table. The fourth model definition rule places this inode in the `RootDirectoryInode` set. The fifth model definition rule states that objects that are not in the `UsedBlock` set should be placed in the `FreeBlock` set.

The `bitmap` array in the `BlockBitmap` object records whether blocks in the file system are in use. The sixth model definition rule uses this array to construct the `BlockStatus` relation to map blocks to boolean values that indicate whether the blocks are in use.

The remaining model definition rules, in order, construct a set of directory entries; decode these entries to construct a set of file inodes; construct the relation `InodeOf`, which maps directory entries to the corresponding inodes; construct the relation `ReferenceCount`, which maps inodes to the corresponding reference counts; decode the inodes to construct the set of blocks in files; and construct the relation `Contents`, which maps inodes to the blocks that store the contents.

In general, we intend that developers will use the sets in the abstract model to group all the objects with the same consistency properties together. For example, the eleventh model definition rule places all of the blocks that store the contents of files in the `FileBlock` set. We intend that developers will use relations to map these objects to primitive values (or other objects) that these objects are conceptually associated with.

```
1. true => d.s in SuperBlock
2. true => d.b[d.s.blockbitmap] as BlockBitmap in BlockBitmap
3. true => d.b[d.s.inodetable] as InodeTable in InodeTable
4. for itb in InodeTableBlock, true => itb.itable[d.s.rootdirectoryinode] in
     RootDirectoryInode
5. for j=0 to d.s.numberofblocks-1, !(d.b[j] in UsedBlock) => d.b[j] in FreeBlock
6. for j=0 to d.s.numberofblocks-1, for bbb in BlockBitmap, true =>
     <d.b[j],bbb.bitmap[j]> in BlockStatus
7. for di in DirectoryInode, for j=0 to (d.s.blocksize/128-1), for k=0 to 11,
     true => (d.b[di.block[k]] as DirectoryBlock).de[j] in DirectoryEntry
8. for e in DirectoryEntry,for itb in InodeTable, e.inode!=0 =>
     itb.itable[e.inode] in FileInode
9. for e in DirectoryEntry,for itb in InodeTable, e.inode!=0 =>
     <e, itb.itable[e.inode]> in InodeOf
10. for j in UsedInode, true => <j,j.referencecount> in ReferenceCount
11. for i in UsedInode, for j=0 to 11, !(i.block[j]=0) =>
     d.b[i.block[j]] in FileBlock
12. for i in UsedInode, for j=0 to 11, !(i.block[j]=0) =>
     <i,d.b[i.block[j]]> in Contents
```

Figure 3-4: Model Definition Rules

3.1.3 Consistency Constraints

Consistency consistency constraints specify the data structure consistency properties that should hold for the abstract model. Consistency constraints are specified using the consistency constraint language. The specification language allows the developer to use the logical connectives (and, or, not) to assemble the body of a constraint out of basic propositions. These basic propositions express basic properties on the sets and relations. Our consistency constraint language includes universal quantifiers that can quantify over the objects in the sets or the tuples in the relations. The consistency constraints in Figure 3-5 identify the consistency properties that the file

system model must satisfy.

The first pair of constraints uses the `size` predicate to specify that the `BlockBitmap` and `InodeTable` sets must contain exactly one object. Because the model definition rules place specific disk structures into these sets, the consistency constraints function to ensure that these disk structures exist.

The next constraints specify properties that all objects in a given set must satisfy. The third constraint ensures that the `BlockStatus` relation maps blocks in the `UsedBlock` set to the boolean value `true`, and the fourth consistency constraint ensures that the `BlockStatus` relation maps blocks in the `FreeBlock` set to the boolean value `false`. The combined effect of these two constraints is to ensure that the `BlockStatus` relation correctly records whether blocks are in use or free.

The fifth consistency constraint `for i in UsedInode, i.ReferenceCount=size(InodeOf.i)` specifies that the reference count for each used inode must reflect the number of directory entries that refer to that inode. [1] The final consistency constraint ensures that each file or directory block is referenced by at most one directory entry. Formally, this constraint ensures that the inverse of the `Contents` relation evaluated on a member of the `FileDirectoryBlock` set contains exactly one object. In general, each consistency constraint is a first-order logical formula consisting of a sequence of quantifiers followed by a quantifier-free boolean formula of basic propositions.

```
1. size(BlockBitmap)=1
2. size(InodeTable)=1
3. for u in UsedBlock, u.BlockStatus=true
4. for f in FreeBlock, f.BlockStatus=false
5. for i in UsedInode, i.ReferenceCount=size(InodeOf.i)
6. for b in FileDirectoryBlock,size(Contents.b)=1
```

Figure 3-5: Consistency Constraints

[1] The expression `InodeOf.i` denotes the image of `i` under the inverse of the `InodeOf` relation — in other words, the set of all objects that `InodeOf` relates to `i`.

3.2 Repair Algorithm

The generated repair algorithm finds violations of the consistency constraints in the model, synthesizes model repairs that eliminate the consistency violations, and then translates model repairs into concrete data structure updates. Because there may be many constraint violations, our repair algorithm then repeats the model construction, the consistency violation detection phase, the model repair phase, and the data structure update phase until all constraints hold.

We illustrate the operation of the repair algorithm by discussing the steps it takes to repair a file system whose superblock has an out-of-bounds bitmap block index. At the end of the model construction process, the repair algorithm constructs the abstract model shown in Figure 3-6.

Notice that the `BlockBitmap` set in Figure 3-6 is empty. This occurs because the bitmap block index `d.s.blockbitmap` is out-of-bounds, and therefore the second model definition rule does not insert any blocks into the `BlockBitmap` set. Since the `BlockBitmap` set is empty, the model violates the first consistency constraint from Figure 3-5.

To repair this violation, the repair algorithm performs a model repair that adds a block from the `FreeBlock` set (the developer specifies this set as the source of new `Blocks` to insert into other sets) to the `BlockBitmap` set. To generate an update that implements this addition, the compiler finds the model definition rule that constructs the `BlockBitmap` set. The relevant model definition rule from Figure 3-4 is `true => d.b[d.s.blockbitmap] as BlockBitmap in BlockBitmap`. The compiler analyzes this model definition rule to determine that the inclusion condition of this rule adds the block from the array `d.b` at offset `d.s.blockbitmap` to the `BlockBitmap` set. As a result, the repair algorithm can make this model definition rule add a block to the `BlockBitmap` set by calculating the block's index in `d.b` and setting `d.s.blockbitmap` equal to this value. Notice that calculating the block's index in `d.b` only works if the selected block is a member of the array `d.b`. To ensure that the selected block is a member of the array `d.b`, it suffices to show that all members of the set from which

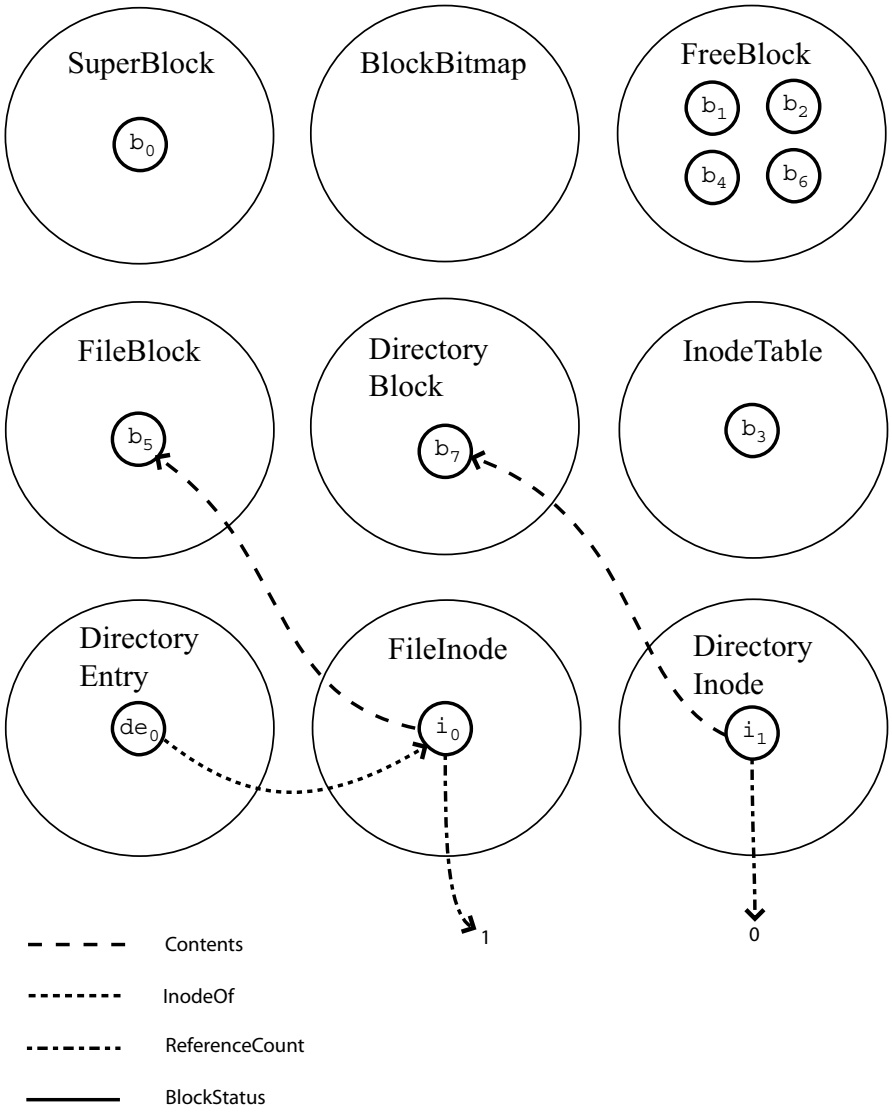


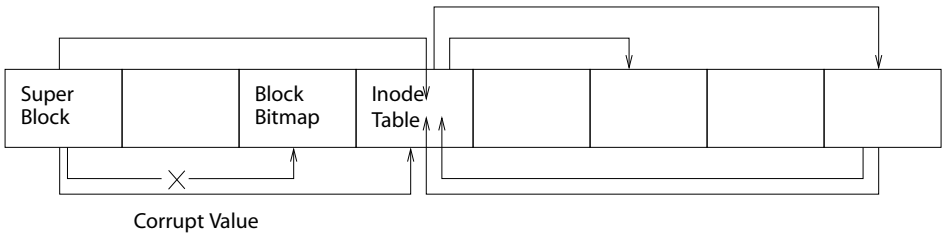
Figure 3-6: Broken Model

the block was selected, the `FreeBlock` set, are in this array. To check this condition, the compiler analyzes the rule that constructs the `FreeBlock` set to determine that all blocks in the `FreeBlock` set are from the array `d.b`, and therefore it can set `d.s.blockbitmap` to the index `j` of the block `d.b[j]`. The resulting file system is shown in part `B` of Figure 3-7. An additional effect of the data structure update is that the block becomes a member of the set `UsedBlock` of used blocks and is removed from the `FreeBlock` set.

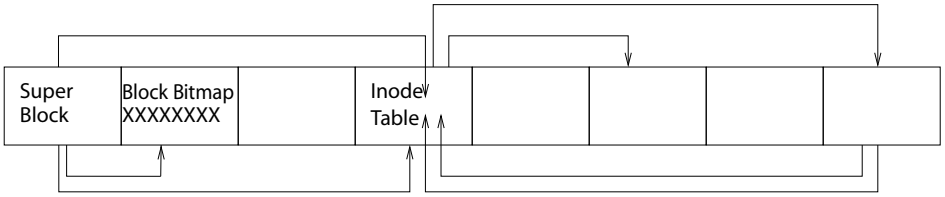
After this update, the repair algorithm rebuilds the abstract model. This rebuilt abstract model is given in Figure 3-8. Notice that although the `BlockStatus` relation now maps blocks to boolean values, it does not correctly map blocks in the `FreeBlock` set to the value `0` (representing false) nor blocks in the `UsedBlock` set to the value `1` (representing true). Therefore, when the repair algorithm checks the consistency constraints, it discovers several violations of the consistency constraints 3 and 4 in Figure 3-5. These violations occur because the bits in the new bitmap block do not correctly reflect which blocks are free and which blocks are in use. The repair algorithm repairs each of these violations by repairing the incorrect tuples in the `BlockStatus` relation to reflect the contents of the `UsedBlock` and `FreeBlock` sets — if a block u is used, the repair algorithm ensures that $\langle u, true \rangle$ (and no other tuple with u as its first component) is in the `BlockStatus` relation, and if a block u is free, the repair algorithm ensures that $\langle u, false \rangle$ is in the `BlockStatus` relation.

The translation of a model repair that removes an object (or tuple) from a set (or relation) to the concrete data structures occurs when the repair algorithm rebuilds the model. For example, whenever model definition rule 6 in Figure 3-4 attempts to add an incorrect tuple to the `BlockStatus` relation, the repair algorithm performs a data structure update that sets the corresponding element (`bbb.bitmap[j]`) in the concrete data structure to the correct value. Part `C` of Figure 3-7 presents the repaired file system after these updates have been performed. After the repair algorithm performs all of these updates, it rebuilds the model. Figure 3-9 shows the rebuilt abstract model. The repair algorithm then checks the consistency constraints on this model and finds that the repaired model satisfies all of the consistency constraints.

A. Initial corrupt file system



B. Allocated Block Bitmap



C. Repaired file system

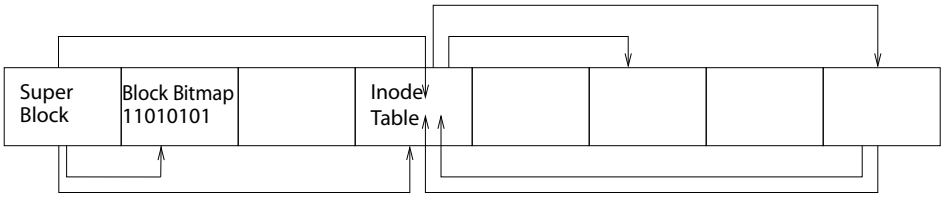


Figure 3-7: Repair Sequence

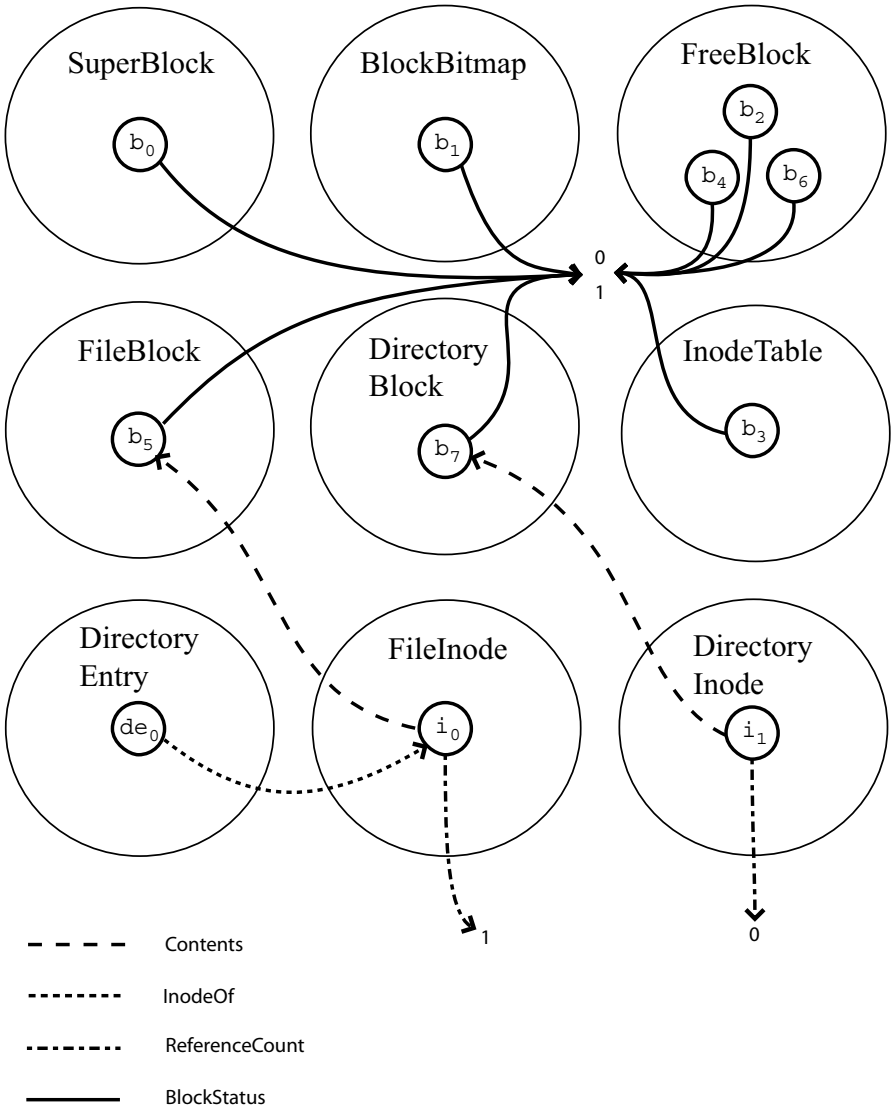


Figure 3-8: Model with a BlockBitmap

Therefore, the repair process is complete and the repair algorithm exits.

In this case, the repair algorithm used the redundant information in the file system to regenerate the bitmap block without losing information. In general, the repair algorithm will produce a consistent data structure that satisfies the consistency constraints and is heuristically close to the original inconsistent data structure. Of course, the new consistent data structure may differ from the data structure that a (hypothetical) correct application would have produced, especially if the inconsistent data structure contains less information.

3.3 Repair Algorithm Generation

As illustrated in the preceding section, a successful repair algorithm must: 1) traverse the model to find any inconsistencies, 2) identify the appropriate model repair action, 3) perform the data structure updates to implement the model repair action, 4) repeat to eliminate any remaining or newly introduced inconsistencies, and 5) terminate. As described above, the algorithm uses goal-directed reasoning to derive the concrete data structure updates from the model definition rules. Goal-directed reasoning enables the repair algorithm to guarantee that the repaired data structures satisfy the consistency specification.

3.4 Repair Dependence Graph

A basic issue in ensuring repair termination is that repairing one constraint may cause the repair algorithm to violate another constraint. If the repair of the newly violated constraint, in turn, causes the originally repaired constraint to become violated, there is an infinite repair loop. The compiler uses a repair dependence graph to reason about termination (see Section 6.5). The edges in this graph capture any invalidation effects that the repair of one constraint may have on other constraints; the absence of cycles in this graph guarantees that all repairs will terminate.

Figure 3-10 presents the repair dependence graph for the example. Although this

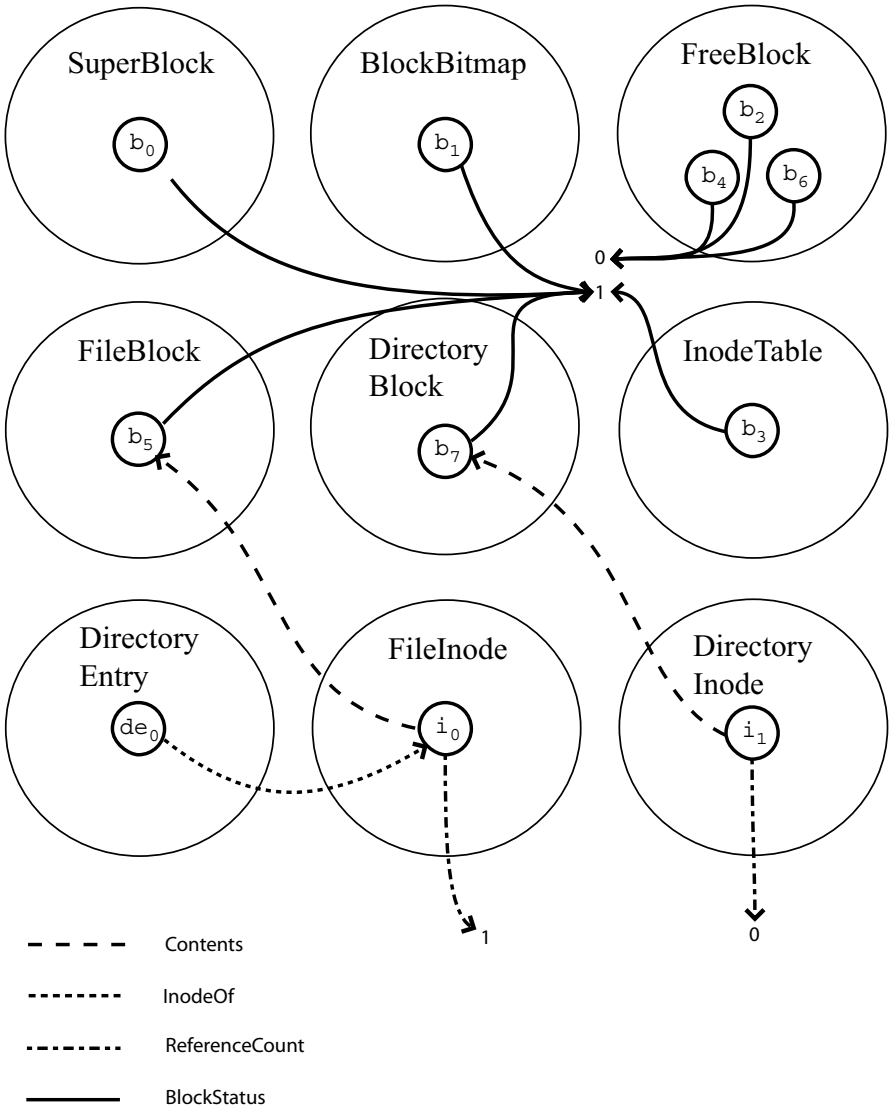


Figure 3-9: Repaired Model

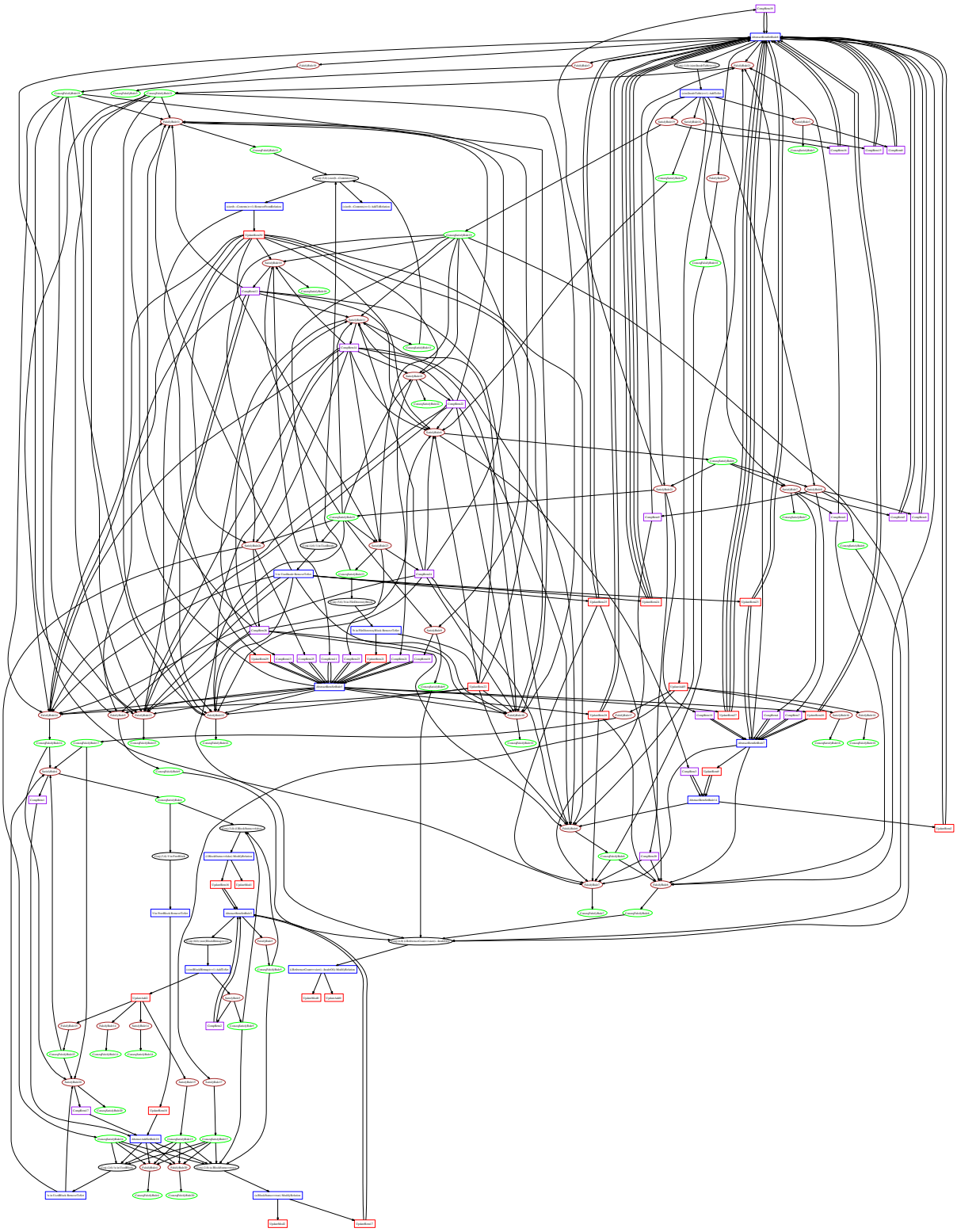


Figure 3-10: Repair Dependence Graph

graph contains many cycles, it is possible to remove these cycles by pruning nodes (and therefore the corresponding repair actions) from the graph provided that other repair actions can be used to satisfy the corresponding constraint (see Section 6.6). This pruning step potentially eliminates desirable repair actions in favor of less desirable repair actions (that potentially delete objects). However, the developer obtains a termination guarantee by pruning the repair actions. We believe that this tradeoff is worthwhile — without a termination guarantee the generated repair algorithm may loop when it is deployed.

Figure 3-11 presents the pruned repair dependence graph for the file system example. Since this graph is acyclic and contains all of the necessary repair actions, the corresponding repair algorithm terminates and successfully repairs violations of all of the constraints. In some cases, it is not possible to both remove the cycles in the graph and retain a sufficient set of repair actions. In this case the specification compiler will fail to generate a repair algorithm. We made a design trade-off here in favor of termination; instead of failing to generate a repair algorithm, the specification compiler could generate repair algorithms that may fail to repair constraint violation or may fail to terminate.

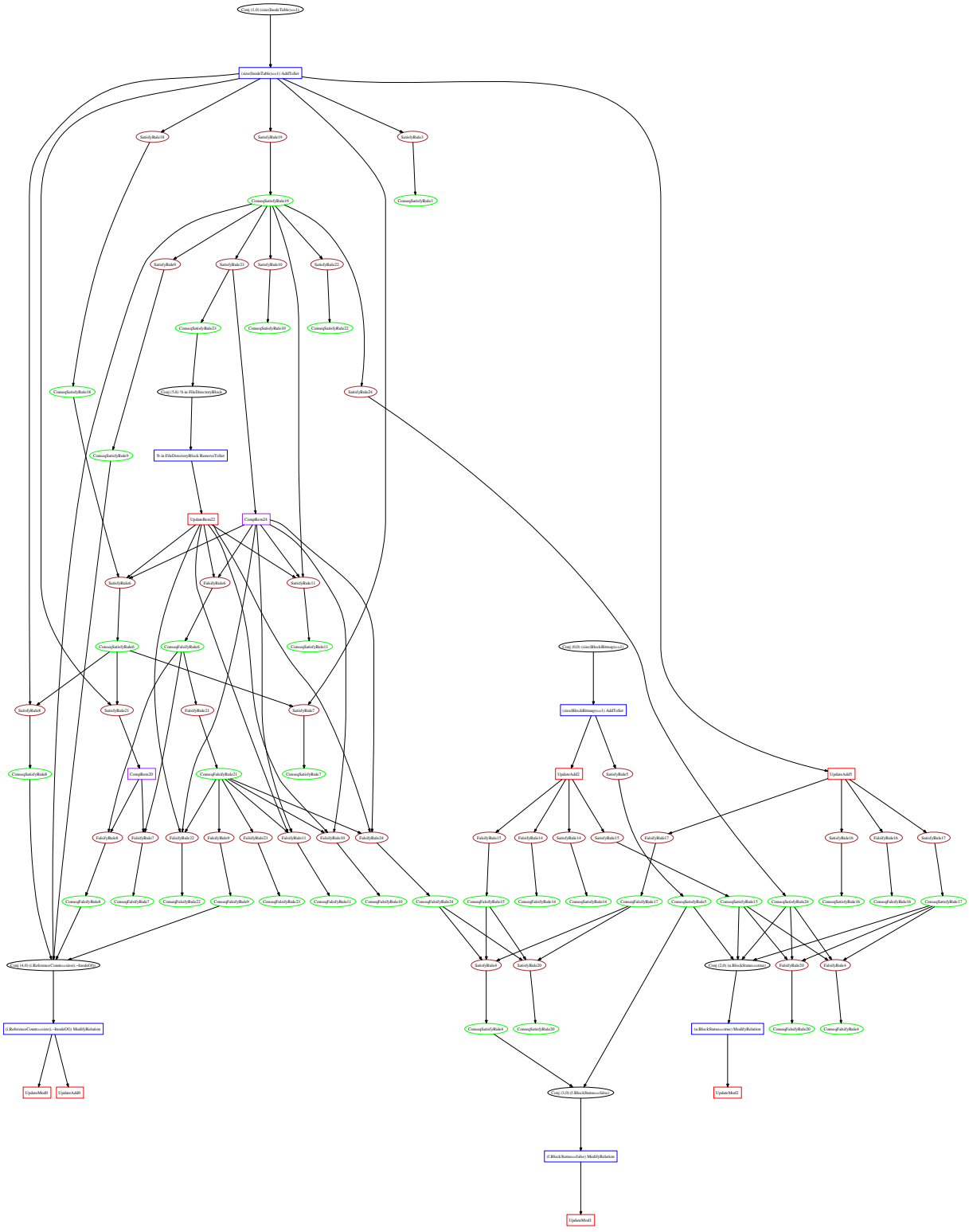


Figure 3-11: Pruned Repair Dependence Graph

Chapter 4

The Specification Language

The goal of our repair tool is to generate repair algorithms that repair damaged data structures. The repair tool uses a consistency specification to generate code that checks whether a data structure is inconsistent and repairs any inconsistencies. The consistency specification consists of three parts: the structure definition, the model definition, and the consistency constraints. This chapter presents the specification languages used by the repair tool.

4.1 Structure Definition Language

Without some way of decoding a data structure, the repair tool can only view a data structure as a string of uninterpretable bits. Our specification compiler uses a structure definition specification to enable the repair tool to decode the strings of bits that represent data structures into objects with fields. The developer uses the structure definition language to declare the layout of the data structures in memory. This structure definition language is similar to the C language with extensions to support variable length arrays, support for packed bit arrays, support for specifying arbitrary alignments, and support for various types of inheritance. Figure 4-1 presents the grammar for this language.

The developer uses this language to declare the fields (or array elements) that comprise the data structure in a similar manner to C structure definitions. This

language allows the developer to declare the following kinds of fields of a structure: 8, 16, and 32 bit integers; structures; pointers to structures; packed bit arrays; arrays of integers; arrays of structures; arrays of pointers to structures; and reserved space.

We have developed a tool to automatically extract the structure definitions from the debugging information in binaries. This removes the burden of writing the structure definition specification from the developer.

One issue with specifying the layout of an object is specifying how the compiler aligns the fields in objects. Many compilers align fields to the nearest machine word. The developer can use the reserved space declaration to ensure that fields are properly aligned. Not that our structure definition extraction tool automatically handles ensuring that the alignment of fields is properly specified. The reserved space declaration can also be used to declare unused space in data structures. This space may be used in future version of the software or by type definitions that inherit from this declaration.

The structure definition language supports two forms of inheritance: extensional inheritance and structural inheritance. Structural inheritance allows a developer to define types that leave a region in the data structure undefined, and then incrementally refine part of that region in a second type definition that declares fields in these undefined parts. The developer declares that one type structurally inherits from another type by using the `subtype of` keyword followed by the type it structurally inherits from in the structure definition. Extensional inheritance allows the developer to define a type that adds fields to a previous type definition. Extensional inheritance was designed to support C++ style inheritance. The developer declares that one type extensionally inherits from another type by using the `subclass of` keyword followed by the type it extensionally inherits from in the structure definition.

Finally, the specification language enables the developer to declare arrays with array bounds that are either constants or expressions over application variables.

$$
\begin{array}{rcl}
structdefn & := & \texttt{structure}\ structurename\ \{fielddefn^*\}\ | \\
 & & \texttt{structure}\ structurename\ \texttt{subtype of}\ structurename\ \{fielddefn^*\}\ | \\
 & & \texttt{structure}\ structurename\ \texttt{subclass of}\ structurename\ \{fielddefn^*\} \\
fielddefn & := & type\ field;\ |\ \texttt{reserved}\ type;\ |\ type\ field[E];\ |\ \texttt{reserved}\ type[E]; \\
type & := & \texttt{boolean}\ |\ \texttt{byte}\ |\ \texttt{short}\ |\ \texttt{int}\ |\ structurename\ |\ structurename\ * \\
E & := & V\ |\ number\ |\ string\ |\ E.field\ |\ E.field[E]\ |\ E-E\ |\ E+E\ |\ E/E\ | \\
 & & E*E
\end{array}
$$

Figure 4-1: Structure Definition Language

4.2 Model Definition Language

The model definition specification specifies how to abstract the concrete data structures. The model definition specification specifies both the sets and relations that comprise the abstract model, and how to construct an abstract model given the corresponding concrete data structure. The specification contains two different sections: the set and relation declarations and the model definition rules. The set and relation declarations declare the sets and relations that comprise the abstract model and state any subsetting and partition constraints between these sets. The model definition rule section specifies the translation of a concrete data structure to an abstract model.

4.2.1 Set and Relation Declarations

The declaration section of the model definition language allows the developer to both declare the sets and relations in the model and to specify subsetting and partition constraints. Figure 4-2 presents the grammar for the set and relation declarations. A set declaration of the form `set S of T` declares a set `S` that contains objects of type `T`, where `T` is either a primitive type or a `struct` type declared in the structure definition part of the specification.

A declaration of the form $\texttt{S partition S}_1, ..., \texttt{S}_\texttt{n}$ declares that the set `S` has `n` subsets $\texttt{S}_1, ..., \texttt{S}_\texttt{n}$ which together partition `S`. Changing the `partition` keyword to `subsets` removes the requirement that the subsets $\texttt{S}_1, ..., \texttt{S}_\texttt{n}$ partition `S` but otherwise

leaves the meaning of the declaration unchanged. The declarations `set` S `of` T : $(S,)^*S$ and `set` S `of` T : `partition` $(S,)^*S$ combine a set declaration and a subset or partition declaration.

A relation declaration of the form `relation R:` $\mathtt{T}_1 -> \mathtt{T}_2$ declares a relation `R` that relates objects of type $\mathtt{T}_1$ to objects of type $\mathtt{T}_2$. A relation declaration of the form `relation R:` $\mathtt{S}_1 -> \mathtt{S}_2$ declares a relation `R` that relates objects in the set $\mathtt{S}_1$ to objects in the set $\mathtt{S}_2$.

$$
\begin{aligned}
D \;:=\; & \texttt{set}\ S\ \texttt{of}\ T \mid S\ \texttt{partition}\ (S,)^*S \mid S\ \texttt{subset}\ (S,)^*S \mid \texttt{set}\ S\ \texttt{of}\ T\texttt{:}\ (S,)^*S \mid \\
& \texttt{set}\ S\ \texttt{of}\ T\texttt{:}\ \texttt{partition}\ (S,)^*S \mid \texttt{relation}\ R\texttt{:}S\texttt{->}S \mid \texttt{relation}\ R\texttt{:}T\texttt{->}T
\end{aligned}
$$

Figure 4-2: Set and Relation Declarations

4.2.2 Model Definition Rules

The model definition rules specify a translation from the concrete data structures into a set- and relation-based abstract model. Each model definition rule has an optional guarded universal quantifier that identifies the scope of the rule, a guard whose predicate must be true for the rule to apply, and an inclusion constraint that specifies either an object that must be in a given set or a tuple that must be in a given relation. Figure 4-3 presents the grammar for the model definition language.

Model definition rule guards use logical connectives (and, or, not) to connect propositions. The model definition language contains three classes of propositions:

- **Numerical Inequalities:** Propositions of the forms FE `=` E, FE `<` E, FE `<=` E, FE `>=` E, and FE `>` E check numerical inequalities on the concrete data structures. Developers can use these propositions to ensure that a field has a legal value before translating the field's state into the abstract model.

- **Pointer Inequalities:** Propositions of the form FE `=` E check pointer equalities on the concrete data structures. Developers can use these propositions to

test whether a pointer is null.

- **Inclusion Constraints:** Propositions of the forms E `in` S and $\langle E, E\rangle$ `in` R check properties of the partially constructed abstract model. Developers can use these propositions to test whether an object is a member of a set.

  The presence of negated inclusion constraints can be problematic. For example, the model definition rule `for x in X, !x in Y => x in Y` defines the set `Y` to contain all objects that are not in `Y`. To prevent such problematic model definition rules, restrictions are placed on the use of inclusion constraints in order to ensure that the abstract model is well-formed. Section 5.3.2 discusses these restrictions.

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

Figure 4-3: Model Definition Language

4.3 Consistency Constraint Language

Figure 4-4 presents the grammar for the model constraint language. Each consistency constraint consists of an optional quantifier Q followed by a body B. The consistency constraint language only supports guarded universal quantifiers, and these quantifiers must appear to the left of the constraint body. We omitted existential quantifiers from the language to simplify the consistency checking and repair processes. These restrictions simplify checking the consistency properties: the constraint checking step simply iterates over all of the constraint bindings to evaluate the consistency constraint.

The body uses logical connectives (and, or, not) to combine basic propositions P. The basic propositions capture the data structure consistency properties in terms of the sets and relations in the abstract model. The language includes four classes of basic propositions:

- **Set Size Propositions:** Basic propositions of the forms `size`(SE)`=`c, `size`(SE)`>=`c, and `size`(SE)`<=`c constrain the sizes of sets. Set expressions SE can either be base sets declared in the set and relation declarations or derived sets defined by the application of relations (or their inverses) to a variable. This type of constraint can be used to ensure that certain data structures exist.

- **Inclusion Propositions:** Basic propositions of the form V `in` SE constrain a variable V to be in a set. As above, a set expression SE can either be a set declared in the model declaration or a set defined by the application of relations (or their inverses) to a variable. These constraints are useful for writing boolean statements about the membership of an object in a set (or a tuple in a relation).

- **Numerical Inequality Propositions:** Basic propositions of the forms VE`=`E, VE`<`E, VE`<=`E, VE`>`E, and VE`>=`E enforce numerical inequalities. These propositions are useful for ensuring integer bound constraints and numerical inequalities between different variables or fields.

- **Pointer Inequality Propositions:** Basic propositions of the form VE`=`E enforce pointer inequalities. These propositions are useful for enforcing structural properties of heap data structures. For example, propositions of the form `x.next.prev` = `x` can be used to ensure that back pointers are consistent with forward pointers in a doubly linked list.

4.3.1 Treatment of Undefined Values

Our system supports numerical constraints between field values by using relations to model these fields. If a relation that is used in a numerical expression does not return

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

Figure 4-4: Consistency Constraint Language

exactly one value, the evaluation of an expression containing the relation can produce surprising results. Consider the consistency constraint `for x in X, x.R.R'>0` where the relation `R` models a (possibly null) pointer and the relation `R'` models an integer field. If `x.R` evaluates to the empty set, then `x.R.R'` is empty and therefore `x.R.R'>0` is undefined. Simply prohibiting expressions that may evaluate to an undefined value is problematic, as they can be used as part of a well-defined constraint such as `for x in X, size(x.R)=0 || x.R.R'>0`. Therefore, we must appropriately handle expressions that may have undefined values.

We treat undefined values in our 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. We extend the arithmetic operations to return the `maybe` value when the evaluation of a relation expression that is used in a numerical expression does not return exactly one value. Figure 4-5 gives the truth table for three-valued logic.

4.3.2 Using Relations as Functions

The primary complication in checking constraints has to do with arithmetic and logical expressions involving relations. Consider, for example, an expression of the

a	b	a or b	a and b	!a
false	false	false	false	true
false	maybe	maybe	false	true
false	true	true	false	true
maybe	false	maybe	false	maybe
maybe	maybe	maybe	maybe	maybe
maybe	true	true	maybe	maybe
true	false	true	false	false
true	maybe	true	maybe	false
true	true	true	true	false

Figure 4-5: Three Value Logic Truth Table

form $V_1.R_1$. Strictly speaking, $V_1.R_1$ is the set of objects in the image of V_1 under R_1, not a single value. Our intention is that developers use these expressions only when the relational image contains a single value. We designed the primitive arithmetic and logical operations to take as input two singleton sets and produce the appropriate singleton set as output. When given a non-singleton set as input, the primitives produce the undefined value.

4.3.3 Restrictions on Basic Propositions

In Section 4.3.1, we presented how the repair algorithm treats undefined values when evaluating boolean expressions of basic propositions. Generating a repair action for a basic proposition that has an undefined value can be complicated: the repair action first has to repair the undefined value and then it can enforce the constraint. We decided to simplify the repair actions by requiring that the developer explicitly write constraints that ensure that the arguments to a basic proposition are well-defined. We then ensure that the generated repair algorithm enforces constraints that ensure that a value is well-defined before evaluating any constraints that use that value. To ensure that these values are well-defined, the system requires the following restrictions on the specifications:

- **Relations on Left-Hand Side of Inequalities are Functions:**

If any relation used in the left hand side of an equation maps an object to the empty set or more than one object, the evaluation of the equation is undefined. To simplify the repair procedure for equations, we require that the developer write explicit constraints on each relation appearing on the left hand side of an equation to ensure that the relation is a function over the domain that the inequality is evaluated on. For example, we require that the predicate $VE = E$ appears only in constraints that also contain the conjunctions `!size`$(V.R_1...R_i) = 1$ for $1 \leq i < n$ in its disjunctive normal form or that the specification contains constraints that ensure `size`$(V.R_1...R_i) = 1$ for $1 \leq i < n$.

- **Relations on Left-Hand Side of Inequalities are Injections:**

  Consider the constraint `for s in S, s.R.R'=s.R''`. If `R` maps two objects from set `S` to the same object, then an update to `R'` to satisfy the constraint for one variable binding may violate the constraint for a second variable binding. To ensure that this does not occur, we place the following restrictions on the specifications: for propositions of the form $VE = E$, where VE is of the form $V.R_1.R_2...R_n$, our system requires that the specification contains a constraint for each R_i except R_n that `for` V `in` $\mathcal{R}(R_i)$, `size`$(R_i.V) <= 1$. [1] These constraints ensure that the relations are injections, ensuring that repairs of this constraint for one variable binding will not violate an instance of the constraint for a different variable binding.

- **Relations used as partial functions are partial functions:** We added restrictions to the specifications to ensure that relations that are used as partial functions are partial functions. These restriction simplify the repair algorithm as they make the well-formedness constraints explicit. For propositions of the form $VE = E$, where VE is of the form $V.R_1.R_2...R_n$, our system requires that the specification contains the constraint on R_n that `for` V `in` $\mathcal{D}(R_n)$, `size`$(V.R_n) <= 1$.

[1] The functions $\mathcal{D}$ and $\mathcal{R}$ map a relation to the corresponding domain and range sets, respectively.

- **Prefixes of non-empty set expressions are non-empty:**

  Consider the constraint `for s in S, size(s.R.R')=1`. If `R` maps an object in `S` to the empty set, the repair algorithm would have to perform a repair action that modifies `R` and then repairs `R'`. To simplify the repair actions for size constraints, our specification compiler requires that developer to explicitly write a constraint that the size of `s.R` is non-zero. This simplifies the repair action for this constraint, it now just needs to repair `R'`.

  Formally, our system requires that specifications that contain constraints of the form $\texttt{size}(SE) = c$, $\texttt{size}(SE) >= c$, $\texttt{!size}(SE) <= c$, or V `in` SE where $SE = V.R_1...R_i$, $i > 1$, and $c > 0$ also contain a constraint of the form $\texttt{size}(SE') = c'$, $\texttt{size}(SE') >= c'$, $\texttt{!size}(SE') <= c'$, or V `in` SE' where $SE' = V.R_1...R_{i-1}$ and $c' > 0$.

- **Relations used in expressions as functions are functions:** Our system requires that each specification containing an expression of the form $E.R$ must also contain another constraint 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.

Our system orders the repair of the constraints to ensure that the repair algorithm repairs any required constraints before the constraints that require them. Note that the repair dependence graph does not restrict the ordering of repair actions. Our specification compiler uses a separate graph to keep track of dependences in the order that constraints are repaired.

4.4 Desugaring Set and Relation Declarations

Set declarations allow the developer to specify that one set is a subset of another, or that a group of sets partitions another set. The specification compiler desugars the partition constraints and subset constraints into consistency constraints and model definition rules. Partition and subset constraints both ensure that one set is a subset of another set. It is possible to have cyclic dependences in these constraints (i.e.

$A \subseteq B$, $B \subseteq C$, and $C \subseteq A$). The solution to such a cyclic dependency is to detect the cyclic dependence, and combine all of sets involved in the cyclic dependence (i.e. $A = B = C$).

The specification compiler first analyzes the partition and subset constraints to find cyclic dependences. If the specification compiler discovers a cyclic dependency, the specification compiler simply merges all sets in the cyclic dependency into a single set. Then the specification compiler removes any constraints that it discovers to be extraneous as a result of merging sets in a cyclic dependency to a single set.

Therefore, we can safely assume that the dependences between partition and subset constraints do not include any cycles. Partition constraints are desugared into consistency constraints that quantify over the partitioned set and contain a conjunction for each partition. This conjunction specifies that the object referenced by the quantified variable is in one partition and not in any of the other partitions. Formally, the specification compiler desugars partition constraints of the form `S partition` $\texttt{S}_1$`,` `...,` $\texttt{S}_\texttt{n}$ into model constraints of the form: `for` V `in` S, (V `in` S_1 `and !`V `in` S_2... `and !`V `in` S_n) `or` (`!`V `in` S_1 `and` V `in` S_2... `and !`V `in` S_n) `or` ...`or` (`!`V `in` S_1 `and !`V `in` S_2... `and` V `in` S_n). While this expansion does result in a quadratic increase in the size of the constraint, we have not found this to be an issue in practice. If this quadratic increase proves to be an issue for some future specifications, we can always expand our consistency constraint language to include a predicate that enforces that an object is a member of exactly one set out of a list of sets.

The subset and partition constraints imply that an object in a set is also contained in the set's superset. We implement this by desugaring as follows: we can ensure that the model respects this property by generating additional model definition rules. Specifically, if a model definition rule adds an object to a subset, we generate another model definition rule to add the object to the superset. Formally, if a subset or partition constraint implies $\forall s, s \in S_1 => s \in S_2$, then the specification compiler creates a new model definition rule 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.

Relation declarations can optionally declare domain and range sets. We desugar

declared domain and range sets into consistency constraints that ensure that the objects that compose the tuples in the relations come from the correct sets. The specification compiler desugars relation declarations of the form `relation` $R : S_1$`->`S_2 into consistency constraints of the form: `for` $\langle V_1, V_2 \rangle$ `in` $R, (V_1$ `in` $S_1)$ `and` $(V_2$ `in` $S_2)$. In some cases, this consistency constraint is not necessary as the compiler is able to analyze the model definition rules to determine that objects in the domain and/or range of the relation are always in the correct set by construction. For example, our algorithm does not generate the consistency constraint for S_1 (or S_2) if for each model definition rule of the form $(Q,)^* \; G \Rightarrow \langle FE_1, FE_2 \rangle$ `in` R there is a model definition rule of the form $(Q,)^* \; G \Rightarrow FE_1$ `in` S_1 (or $(Q,)^* \; G \Rightarrow FE_2$ `in` S_2).

4.5 Language Design Rationale

Software developers have used object modelling languages to express object-oriented design properties of software. The design of our specification language was partly inspired by features of object modelling languages such as Alloy [22]. [2] However, there are significant differences between the goals of object modelling and data structure repair that led us to develop a new specification language. Our design rationale for the specification language was guided by the need to balance the following goals of data structure repair: 1) the specification language needs to be simple enough to enable efficient generation of repairs, 2) the specification language needs to be expressive enough to capture important data structure invariants, 3) the specification language must establish a direct connection to the data structures in the application, and 4) the specification language needs to be simple for developers to use.

4.5.1 Set- and Relation-Based Model

We included a set and relation based abstract model in the specification language because it allows the developer to specify sets of objects in the application's heap that

[2] UML [32] also contains some of these features.

should satisfy a given consistency property and provides a means to manage complexity arising from low-level data structure implementation details. The abstraction mechanism gives the specification developer a degree of freedom in how he or she chooses to represent a concrete data structure in the abstract model. This representation freedom enables the system to use a simple consistency constraint language and still be able to capture important data structure invariants. In addition, simplifying the consistency constraint language simplifies both generating repair actions and reasoning about the interactions between repair actions and model constraints.

4.5.2 Model Definition Rules

While our consistency properties are expressed in terms of a set- and relation-based abstract model, applications store their data structures as a string of bits in the heap. Our repair algorithm has to incorporate some mechanism of translating this string of bits into a set- and relation-based abstract model.

One straightforward way of performing this translation is to have one set for each type of object and one relation for each field in each type of object. The translation step would then place all objects in the heap of a given type into the corresponding set, and store the values of fields in the corresponding relations. There are two problems with this approach: this approach requires that the language runtime is able to distinguish different types of objects, and it does not allow constraints to easily differentiate between objects of the same type that participate in different data structures.

The TestEra [27] system performs a similar translation between Java data structures and Alloy [22] models by requiring users to write Java code to perform the translation. The problem with this approach is that the repair algorithm could generate abstract models that do not correspond to any data structure. The user provided code would then fail to translate this abstract model back into a concrete data structure.

We chose to use our model definition rule based approach because it allows the developer to chose how to classify objects, we can analyze the model definition rules

so that the specification compiler can automatically translate model repair actions into data structure updates, and the specification compiler can conservatively reason about the effects of the data structure updates on the model.

4.5.3 Consistency Consistency Constraints

We chose a simplistic set of basic propositions for the model consistency constraints and restricted the use of quantifiers to simplify the evaluation and repair of the constraints. We only allow universal quantifiers that appear to the left of the constraint body, so that the generated repair algorithms can simply use a simplistic consistency checking algorithm that evaluates the constraint body for all quantifier bindings. This evaluation strategy gives the repair algorithm quantifier bindings for any constraint violations, which greatly simplifies the repair algorithm.

4.6 Discussion

We designed the specification language to incorporate an abstract model of the data structures to allow the developer to specify which objects in the heap a constraint should apply to and to abstract low-level details out of consistency constraints. Our current implementation explicitly separates the model definition rules from the consistency constraints in order to give the developer a simpler mental model of the repair process. This explicit separation has a downside, however. Although the set- and relation-based abstract model simplifies many specifications, it can complicate the specification of some simple low-level consistency properties. This occurs because the developer has to provide model definition rules to translate all of the state needed by a consistency property into the abstract model.

Specification languages that allow the consistency properties to reference both the abstract and concrete state would enable the developer to take advantage of the abstract model when it is beneficial and avoid the overhead of the abstract model when it is not. A secondary benefit of this approach is that the developer would likely write specifications which generate smaller abstract models, and such models would require

less memory and time to construct at runtime. This feature should be included in any future implementations of the repair tool that are intended to be deployed as it greatly simplifies writing many common consistency properties. While this feature may complicate the specification compiler, the feature can be implemented using the technique we have developed for satisfying guards of model definition rules.

Chapter 5

The Repair Algorithm

Now that we have presented our specification language, which describes properties of consistent data structures, we are ready to describe how our repair algorithms actually modify data structures to enforce consistency properties. Our specification compiler analyzes the consistency specification to automatically generate a repair algorithm. The generated repair algorithms all use the same basic strategy: construct a model of the data structure, evaluate the consistency properties on this model, reason about repair actions in terms of the model, and then translate the model repair actions into data structure updates. This section discusses the generated repair algorithms.

5.1 Basic Concepts

This section introduces some of the basic concepts in our repair algorithms.

Abstract Model: In our system, the developer specifies consistency properties for an abstract model of the data structures in the software system. This abstract model consists of sets of objects and relations between these objects.

Model Construction: The model construction step of the repair algorithm creates an abstract model that corresponds to the current data structure in the software system. This translation step allows the consistency properties to be evaluated on

the data structures.

Model Repair: Our repair algorithm reasons about repairs in terms of operations on an abstract model. For violated constraints, it generates abstract repairs that modify the state of the abstract model by adding or removing the appropriate object (or tuple) to or from a set (or relation).

Data Structure Update: For each model repair that the repair algorithm may perform, the specification compiler uses goal-directed reasoning to generate a set of data structure updates that implement the model repair. The specification compiler generates code to perform the concrete data structure update by finding the model definition rules that add objects (or tuples) to the set (or relation), and then uses goal-directed reasoning to determine the updates to cause the rule's inclusion condition to add the object (or tuple) to the set (or relation) referenced by the inclusion condition, to ensure that the rule's guard is either satisfied or falsified as required by the model repair, and to ensure that the necessary objects or tuples are in the sets or relations that are quantified over.

Additional Effects: Although data structure updates are intended to implement a specific model repair, they can have additional effects. These effects occur because a data structure update may modify the state accessed by other model definition rules (or even a different binding of the same model definition rule). The result is that a data structure update may change the abstract model in unintended ways. We refer to these changes as additional effects.

Compensation: In some cases, it is desirable to synthesize larger, more precise data structure updates. Our specification compiler generates additional compensation updates to counteract any undesired effects of a data structure update. The combination of the data structure update and these compensation updates can be viewed as a larger, more precise data structure update that implements the desired model repair action.

5.2 Repair Algorithm Overview

Our specification compiler compiles data structure specifications into repair algorithms that check for inconsistencies in a data structure and repair any inconsistencies. The generated repair algorithms must address the following issues: 1) repair algorithms must be able to repair violations of all the constraints in the specifications, 2) the repair algorithms must make repairs to the concrete data structure to satisfy constraints that are expressed in terms of an abstract model, 3) repair algorithms must terminate, and 4) repair algorithms must repair arbitrarily broken data structures.

Our compiler generates repair algorithms that use the following basic repair strategy:

1. **Initial Model Construction:** The repair algorithm first constructs an abstract model as described in Section 5.3.

2. **Inconsistency Detection:** The repair algorithm evaluates the consistency constraints. If the repair algorithm finds a violation, it proceeds to the next step. Otherwise, the data structure is consistent and the repair process exits.

3. **Conjunction Selection:** The repair algorithm selects one of the conjunctions in the disjunctive normal form of the violated constraint. It will ensure that the constraint holds by repairing the basic propositions in this conjunction. The conjunction choice can be controlled by the developer or by a cost function that assigns a cost to the repair of each basic proposition.

4. **Model Repair:** For each violated basic proposition in the conjunction, the repair algorithm performs an abstract repair on the model. These model repairs either add or remove objects (or tuples) from sets (or relations) to satisfy the violated basic propositions.

5. **Data Structure Updates:** The repair algorithm must perform data structure updates to implement the model repair. If the repair algorithm performs

a model repair that adds an object (or a tuple) to a set (or a relation), it immediately performs the corresponding data structure update. If an update removes an object (or tuple) from a set (or a relation) or atomically modifies a relation, the repair algorithm records that this data structure update should be performed when the model is rebuilt. Delaying these data structure updates enables the specification compiler to generate data structure updates without knowing the specific quantifier bindings for the model definition rules. Step 6 performs any delayed data structure updates as it rebuilds the model.

Section 5.5.2 describes how the compiler uses goal-directed reasoning to generate a set of data structure updates to implement a 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) was not in the set (or relation), the repair algorithm checks if a specific data structure update has been recorded for the given object (or tuple) and set (or relation). If one has, the repair algorithm performs that 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 the repair algorithm performs any updates, it recomputes the model. Once the repair algorithm completes this recomputation, it deletes the old model and deletes the updates recorded for objects or tuples. Then it proceeds to Step 2.

5.3 Model Construction

The model definition rules specify a translation from the concrete data structures to an abstract model. The model construction phase constructs the abstract model by evaluating 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 data structures are physically well-formed (i.e. that they reside in allocated memory and do not illegally overlap).

5.3.1 Denotational Semantics

Figure 5-1 gives the denotational semantics of a single model definition rule M. A model m is a mapping from set names and relation names to the corresponding sets of objects or relations between objects. The denotational semantics represents this mapping using a set of tuples. We define the notation $m(S)$ to be the contents of the set or relation S, formally the set $\{v \mid \langle S, v\rangle \in m\}$. The set h models the heap in the running application 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 M to m in the context of h and l. Note that l provides the values of both the variables that the rules use to reference the concrete data structures and the variables bound in the quantifiers.

5.3.2 Negation and the Rule Dependence Graph

Sometimes it is useful to construct a set of objects that do not satisfy some property. For example, a developer may define an active set of objects that participate in a given data structure, and then use negation to specify a free set that contains objects that are not in the active set. Negation complicates model construction because it may introduce a non-monotonicity into the fixed point computation that constructs the relational model.

To address this issue, we allow negation only when a model definition rule's negated inclusion constraint does not depend directly on the set or relation constructed by that model definition rule or indirectly on this set or relation through the

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

$$
\begin{array}{l}
\mathcal{R}[\texttt{for } V \texttt{ in } S, M]\ h\ l\ m\ = \bigcup_{v \in m(S)} \mathcal{R}[M]\ h\ l[V \mapsto v]\ m \\
\mathcal{R}[\texttt{for } \langle V_1, V_2 \rangle \texttt{ in } R, M]\ h\ l\ m\ = \bigcup_{\langle v_1, v_2 \rangle \in m(R)} \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\ = \bigcup_{v=\mathcal{E}[E_1]h\ l\ m}^{\mathcal{E}[E_2]h\ l\ m} \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\ = 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}[number]\ h\ l\ m\ = number \\
\mathcal{E}[string]\ h\ l\ m\ = string \\
\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}[FE.field]\ h\ l\ m\ = b.\langle \mathcal{SE}[FE]\ h\ l\ m, field, b \rangle \in h \\
\mathcal{SE}[FE.field[E]]\ h\ l\ m\ = b.\langle \mathcal{SE}[FE]\ h\ l\ m, field, \mathcal{E}[E]h\ l\ m, b \rangle \in h
\end{array}
$$

Figure 5-1: Denotational Semantics for the Model Definition Language

actions of other model definition rules. For a given model definition rule with negated dependences on sets and/or relations, this restriction allows the model construction algorithm to completely construct those sets and/or relations before performing the fixed-point computation that evaluates the given model definition rule. Because these sets and/or relations are completely constructed, negation of inclusion constraints that reference them does not affect the fixed point computation.

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 adds objects or tuples to 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 in the disjunctive normal form of its guard, 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.

5.3.3 Ensuring a Finite Model

The presence of integers in the abstract model makes it possible to construct infinite models. Consider the pair of model definition rules `true=>0 in S` and `for s in S, true => s+1 in S`. This pair of model definition rules constructs a set `S` that contains all non-negative integers. Unfortunately, the explicit construction of the non-negative integers does not terminate.

We address this complication by placing restrictions on the use of arithmetic with quantified variables. In particular, we construct a second type of 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 adds objects or tuples to a set or relation used in the quantifiers of the second rule. If a model definition rule involved in a cycle contains an inclusion

constraint that contains at least one arithmetic operation and an integer quantified variable that is not used as an array index, the set of model definition rules may construct an infinite model and we reject it. While this restriction is conservative, it has not presented any problems in our experience.

5.3.4 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 pointers that reference unallocated memory or pointers that overlap with other objects. We next describe how we extend the model construction algorithm to deal with these 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 the model construction algorithm 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 is deterministic.

We coded our model construction algorithm 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.

For safe languages such as Java, these low-level safety checks appear to be unnecessary, because it is not possible to write code in Java that produces these low-level errors. However, if a developer is concerned with possible hardware errors, maliciously introduced errors [17], virtual machine errors, or errors in native code these checks would still be useful even for safe languages.

5.4 Consistency Checking

Once our tool has constructed an abstract model, it executes the consistency checking algorithm on this model. For each model constraint, the checking algorithm iterates through the legal quantifier bindings. The checking algorithm then evaluates the constraints according to the semantics presented in Figure 5-2. If the consistency checking algorithm finds a constraint violation, the repair algorithm performs repairs as described in Section 5.5.

In our experience, consistency constraints sometimes contain expressions that do not depend on all of the quantified variables. A naive implementation would re-evaluate these expressions for each quantifier binding. Our specification compiler implements the standard loop invariant hoisting optimization. When the specification compiler determines that an expression does not depend on a quantified variable, it lifts the evaluation of that expression outside of the given quantifier evaluation. This optimization corresponds to loop-invariant code motion.

5.5 Repairing a Single Constraint

The inconsistency detection algorithm iterates over all values of the quantified variables in the consistency constraints, evaluating the body of the constraint for each possible combination of the values. If the body evaluates to false, the algorithm has

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

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

Figure 5-2: Denotational Semantics for Consistency Constraints

detected a violation and has computed an explicit set of bindings for the quantified variables that causes the constraint body to evaluate to false. The compiler converts the constraint to disjunctive normal form (disjunctions of conjunctions of basic propositions), and performs steps 3 through 5 in the repair algorithm description from Section 5.2.

- **Conjunction or Quantifier Selection:** Satisfying all of the basic propositions in any of the constraint's conjunctions will ensure that the constraint is satisfied. Alternatively, the repair algorithm may remove an object (or tuple) from a set (or relation) that the constraint is quantified over to eliminate the quantifier binding that makes the constraint false. The first step is therefore to either select a conjunction to satisfy or an object (or tuple) to remove from a set (or relation) that the constraint quantifies over.

- **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:** The repair algorithm uses a set of data structure updates to implement each model repair action; the model definition rules determine the specific set of updates.

- **Compensation:** Data structure updates may have additional effects beyond the desired model repair that cause the model to change in undesirable ways, specifically by adding objects to sets or tuples to relations. It is sometimes possible to generate more precise updates that prevent these additional effects by performing additional compensation updates that falsify the guards in the model definition rules that caused the additions to take place.

At this point, the algorithm has repaired a particular violated constraint. However, some constraints may still be violated and the data structure updates may have violated additional constraints. The repair algorithm therefore rebuilds the model and repairs any new or remaining violated constraints.

5.5.1 Model Repair Actions

The model repair action taken to repair a violated basic proposition depends on the form of the proposition. The generated repair algorithm performs the following model repairs for the basic propositions:

- **Size Propositions:** For size propositions, such as `size(BlockBitmap)=1`, the generated repair algorithm simply adds or removes the minimal number of objects (or tuples) to or from the appropriate set (or relation) necessary to satisfy the proposition.

- **Inequalities:** For inequality propositions, such as `i.ReferenceCount=size(InodeOf.i)`, the generated repair algorithm computes the right hand side of the inequality, adds or subtracts one if the comparison is a greater than or less than comparison, then assigns this value to the left hand side of the inequality. For $!=$ the specification compiler currently adds one to the right hand side and assigns this value to the left hand side. Note that the compiler rewrites the comparison operation of a negated inequality to remove the negation.

  In general, inequalities can be solved by modifying the relations that appear on the right hand side. Our specification compiler does not currently generate such repair actions. However, the user can always rewrite the constraint so that these relations appear on the left hand side instead.

- **Inclusion Propositions:** To make an inclusion proposition true, the generated repair algorithm adds the specified object (or tuple) to the specified set (or relation). To make an inclusion proposition false, the generated repair algorithm removes the specified object (or tuple) from the specified set (or relation).

5.5.2 Data Structure Updates

We next discuss how the compiler uses goal-directed reasoning to translate model repairs into actions that correctly update the concrete data structures. Given a

model repair that adds an object (or tuple) to a set (or relation), the compiler finds all model definition rules that contain 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 normalize the guards to disjunctive normal form. The compiler selects a model definition rule, matches the inclusion condition in the model definition rule to the object (or tuple) to be added, selects one of the guards' conjunctions, then generates updates to the data structure that ensure that all of the propositions in the conjunction are true and that the model definition rule's inclusion condition is equal to the object (or tuple) to be added. The specific update depends on the form of the proposition; e.g. for inequality propositions such as `v.f` $< E$, the update computes E, subtracts one from E to generate a value that satisfies the proposition, then assigns this value to `v.f`.

In order to generate an update using a model definition rule, the compiler needs to know what objects the quantified variables should be bound to. The compiler resolves these bindings using one of two different strategies:

- If a set (or relation) that the model definition rules quantifies over contains at most one object (or tuple) then the corresponding variable binding is trivially equal to this object (or tuple). [1]

- If any of the expressions in the inclusion condition of the model definition rules is a variable, then that variable is bound to the object (or the appropriate half of the tuple) to be added to the set (or relation). The compiler may need to generate additional model repairs that ensure that the objects are included in the appropriate set.

Finally, it is possible that one operation in an update changes state that is referenced by another operation in the update. The specification compiler handles this

[1] If there is no other constraint that the given set (or relation) must contain this element, the update may have to perform an abstract repair that adds an object (or tuple) to the set (or relation).

issue by constructing a dependence graph between the various pieces of an update. The compiler then topologically sorts the pieces of updates. If this is not possible, the specification compiler rejects the update. Otherwise, the specification compiler generates the update in topological order to ensure that one piece of the update will not falsify another piece.

The compiler uses a similar strategy to implement repairs that remove an object (or tuple) from a set (or relation) with one major simplification: it is not necessary to compute bindings for the quantified variables for a removal. Instead, the generated repair code can simply keep track of what quantifier bindings result in a model definition rule adding an object (or tuple) to a set (or relation). To implement the repair, the specification compiler chooses a set of propositions that includes at least one proposition from each conjunction of each rule that could cause the object (or tuple) to appear in the set (or relation). It then generates actions that falsify the propositions in this set. The compiler statically verifies that these sets of propositions will not be contradictory. Finally, the compiler checks that there is no dependence cycle between propositions that use and define the same field or variable. The compiler generates a data structure update that satisfies the corresponding set of propositions in a dependence-preserving order.

5.5.3 Atomic Modifications

Consider, for example, a model definition rule of the form `true => <v, v.f> in R`. No possible action can just remove the tuple `<v,v.f>` from `R` because the rule does not have a guard that can be falsified or quantify over a set or relation from which the repair algorithm could remove an object or tuple. In this case, if the presence of the tuple with the old value of `v.f` causes the model to be inconsistent, the repair algorithm performs an update that atomically replaces the old value of the tuple with a new value.

5.5.4 Recursive Data Structures

Trees and linked lists are commonly implemented using 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 that adds the head of the recursive data structure to S and a recursive case model definition rule of the form `for` V `in` $S, G' \Rightarrow FE'$ `in` S that adds the rest of the recursive data structure to S. The repair algorithm can consider two cases for adding an object to a recursive data structure: either it can make the object the head of the recursive data structure or it can add the object after another object in the recursive data structure.

If the repair algorithm makes the object O the head of the data structure, the repair algorithm must make base case model definition rule add O to the set S. To do this the repair algorithm must make the guard G of this model definition rule true and must make the expression FE inclusion condition equal to O, i.e. it must make $FE = O$ true. To avoid losing elements in the recursive data structure, the repair action must use the recursive case model definition rule to ensure that the original head object is still in S. The repair algorithm ensures this by binding the quantified variable V in the recursive case model definition rule to the head object O, then making the guard $G'[V/O]$ [2] of this model definition rule true and making the expression in the inclusion condition $FE'[V/O]$ equal to the original head O', i.e. making $FE'[V/O] = O'$ true. If the recursive data structure was originally empty, i.e. G was initially false, we want to ensure that the recursive case model definition rule does not add any object to the set S. The repair algorithm does this by making its guard false, i.e. making $\neg G'[V/O]$ true.

If the repair algorithm adds the object O after the object O', the repair algorithm must make recursive case model definition rule for the binding of the variable V to O' add O to the set S. To do this the repair algorithm must make the guard $G'[V/O']$ of this model definition rule true and must make the expression $FE'_{V/O'}$ inclusion condition equal to O , i.e. it must make $FE = O$ true. To avoid losing elements

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

in the recursive data structure, the repair action must also use the recursive case model definition rule to ensure that the original object that followed O' is still in S. The repair algorithm does this by binding the variable V in the recursive case model definition rule to the new object O, then making the guard $G'[V/O]$ of this model definition rule true and making the expression in the inclusion condition $FE'[V/O]$ equal to the object O'', i.e. making $FE'[V/O] = O''$ true. If the O' was originally the tail of the recursive data structure, i.e. $G'[V/O']$ was initially false, we want to ensure that the recursive case model definition rule does not add any object to the set S. The repair algorithm does this by making its guard false, i.e. making $\neg G'[V/O]$ true.

We use similar reasoning to generate operations that remove objects from recursive data structures. 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 affect 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.

5.5.5 Compensation Updates

Consider the pair of model definition rules `ptr != null => ptr in AllInts` and `(ptr != null) and (ptr.value >= 0) => ptr in NonNegInts` from the example in Chapter 2. Suppose that the specification compiler needs to generate a data

structure update to implement a model repair that adds a newly allocated object to the `AllInts` set. The use of goal-directed reasoning generates an update that sets the variable `ptr` equal to the address of the newly allocated object. However, depending on the newly allocated object's `value` field, this object may also be added to the `NonNegInts` set. Note that the desired model repair does not add the object to this set, and in some cases, this additional effect in the concrete data structure update may violate consistency constraints.

Consider an update that that sets the `value` field of the newly allocated object to -1 if the object was going to be added to the `NonNegInts` set. Note that this update would prevent the additional effect from occurring. We call such an update a compensation update. Note that the combination of the original data structure update and the compensation update more precisely implements the desired model repair.

Therefore, we augment our translation algorithm to analyze the model definition rules to, when possible, automatically generate additional compensation updates to eliminate the additional effects. When a model definition rule may be affected by a data structure update, our algorithm examines that rule to derive additional updates that restore its original value. The net effect is to improve the precision of the translation by synthesizing larger, more precise data structure updates for each model repair.

There are two strategies to prevent an increase in the scope of a model definition rule:

- One strategy is to make the guard of the model definition rule false when the scope of the model definition rule increases. This will prevent the model definition rule from adding the new object (or tuple) to the set (or relation). The specification compiler generates such compensation updates by negating the model definition rule's guard, converting the negated guard into disjunctive normal form, and then selecting a conjunction to make true.

- A second strategy is to prevent the model definition rule from increasing scope

by removing the objects (or tuples) that the quantified variables are bound to when the rule increase scope. To implement this strategy, the specification compiler generates a model repair action that removes the object (or tuple) that the quantified variable is bound to from the set (or relation) that guards the quantifier.

5.5.6 New Objects

A 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. For example, repairing a violation of the constraint `size(BlockBitmap)=1` caused by the `BlockBitmap` set being empty requires the repair algorithm to find a block to place in this set. Other sets (as specified in the set and relation definition) are one potential source for new objects. For primitive types, such as integers, the action can simply synthesize new values. For `struct`s, memory allocation primitives are a potential source of new objects. Our tool allows the developer to specify on a per-set basis a set where the repair algorithm can obtain new objects from or a memory allocation method that the repair algorithm can call. Otherwise, our tool defaults to using the standard malloc memory allocator.

Chapter 6

The Repair Dependence Graph

We have presented our core repair algorithm. However, we have not yet discussed how our specification compiler determines that a generated repair algorithm terminates. The specification compiler constructs a *repair dependence graph* $\langle N, E \rangle$ to reason about the termination of the repair algorithm. Nodes in this graph represent conjunctions in the DNF of the consistency constraints, repair actions, and model definition rules. One node has a dependence on a second node if the repair compiler may be required to performed the event represented by first node as a result of the event represented by the second node, or if the event represented by the first node may occurs as a result of the event represented by the second node. For example, we say that a data structure update depends on the corresponding model repair, because the repair algorithm may have to perform the data structure update to implement the model repair. Note that events in an individual repair follow paths on the graph. For example, the repair algorithm decides to repair a conjunction; then it performs a model repair; then it implements this model repair by performing the corresponding data structure update, which may change the scope of model definition rules; and finally the repair algorithm may perform compensation updates to counteract these scope changes. These chains of events corresponds to the paths on the repair dependence graph that start from the model conjunction.

Edges capture dependences between the consistency constraints, repair actions, model definition rules, and choices in the repair process. In particular, an edge may

denote that the repair of a constraint requires a given model repair, that the implementation of a model repair requires a given data structure update, that performing a repair action may affect what objects (or tuples) model definition rules add to sets (or relations), or that a repair action or change in a model definition rule's scope may violate a constraint. The absence of cycles in the repair dependence graph ensures that the corresponding repair algorithm will not perform any infinite repair sequences and therefore terminates. Section 6.5 presents a correctness argument that the absence of cycles in the repair dependence graph implies termination. Figure 6-1 is a schema of the types of nodes and edges in the repair dependence graphs. Sections 6.1 and 6.2 discuss the nodes and edges in this figure.

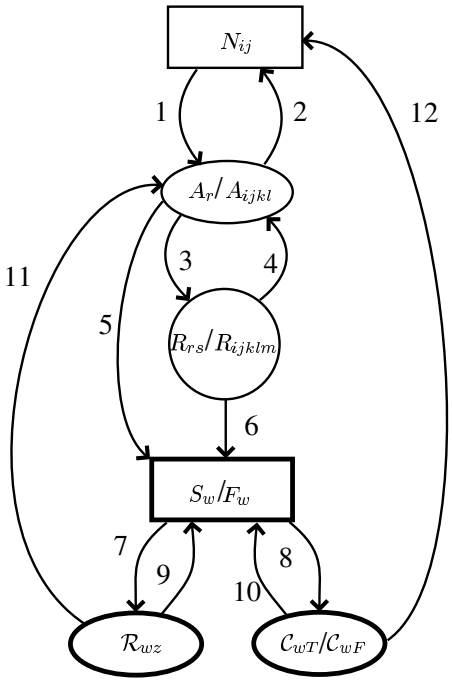


Figure 6-1: Repair Dependence Graph Schema

6.1 Nodes in Repair Dependence Graph

The repair dependence graph contains the following classes of nodes:

- **Model conjunction nodes (rectangle labeled N_{ij}):** In disjunctive normal form, each consistency 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 consistency constraint C_i and an additional node $N_{ij'}$, where $j' = j_{max} + l$, for each quantifier Q_{il} in the consistency constraint.

- **Model repair nodes (ellipse labeled A_{ijkl}/A_r):** 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 to model insertions, and the other to model removals.

- **Data structure update nodes (circle labeled R_{rs}/R_{ijklm}):** 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.

- **Scope increase and decrease nodes (bold rectangle labeled S_w/F_w):** For each model definition rule M_w, there is an scope increase node S_w and a scope decrease node F_w. These nodes represent the additional effects that a data structure update may have 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 (bold ellipses labeled $\mathcal{C}_{wT}/\mathcal{C}_{wF}$, or $\mathcal{R}_{wz}$):** For each model definition rule M_w, there is a pair of rule consequence nodes $\mathcal{C}_{wT}$ and $\mathcal{C}_{wF}$ that 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 $\bigcup_z \{\mathcal{R}_{wz}\}$ that represent compensation

updates that the repair algorithm may use to prevent undesired scope increases of a model definition rule.

6.2 Edges in the Graph

The edges E in the repair dependence graph represent how model and data structure repairs may affect other parts of the model and data structures. These edges capture dependences between repaired conjunctions, model repairs, data structure updates, and model definition rules. The graph contains the following classes of edges (ordered by edge number in Figure 6-1):

1. **Edges from model conjunction nodes to model repair nodes:** Repairing a constraint by making a model conjunction true may require the repair algorithm to perform model repairs. These edges capture this dependence.

   There is an edge $\langle N_{ij}, A_{ijkl} \rangle \in E$ from each model conjunction node N_{ij} to each model repair node A_{ijkl} that may repair one of the basic propositions in the conjunction C_{ij}.

2. **Edges from model repair nodes to model conjunction nodes:** Performing a model repair may make a conjunction false, potentially requiring the repair of a consistency constraint. These edges capture this dependence.

   There is an edge $\langle A_{ijkl}, N_{i'j'} \rangle \in E$ (or $\langle A_r, N_{i'j'} \rangle \in E$) if the repair corresponding to A_{ijkl} (or A_r) may falsify the conjunction $C_{i'j'}$ or expand the quantifier scope of the constraint containing $C_{i'j'}$. The graph construction algorithm uses the conditions discussed below in Section 6.3.1 to compute whether or not the repair may falsify the conjunction.

3. **Edges from model repair nodes to data structure update nodes:** Implementing a model repair on the concrete data structure requires the repair algorithm to perform the corresponding concrete data structure update. These edges capture this dependence.

There is an edge $\langle A_{ijkl}, R_{ijklm} \rangle \in E$ or $\langle A_r, R_{rs} \rangle \in E$ for each model repair node A_{ijkl} or A_r and data structure update node R_{ijklm} or R_{rs} that implements the repair.

4. **Edges from data structure update nodes to model repair nodes:** Data structure updates are designed to make a model definition rule add or remove an object (or tuple) to or from a set (or relation). Making a model definition rule add or remove an object (or tuple) to or from a set (or relation) may require making an inclusion constraint in the model definition rule true or false, or adding or removing an object (or tuple) to or from a set (or relation) that the model definition rule quantifies over. As a result, the repair algorithm may need to perform a model repair to implement a data structure update. These edges capture this dependence.

   There is an edge $\langle R_{ijklm}, A_r \rangle \in E$ (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.

5. **Edges from model repair nodes to increase or decrease scope nodes:** The changes that a model repair makes to the relational model may cause model definition rules to add or remove additional objects (or tuples) from sets (or relations). These edges capture this dependence.

   The following edges characterize how model repairs may increase or decrease the scope of quantifiers of model definition rules. There is an edge $\langle A_{ijkl}, S_w \rangle \in E$ (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. There is an edge $\langle A_{ijkl}, F_w \rangle \in E$ (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.

6. **Edges from data structure update nodes to increase or decrease scope nodes:** A data structure update may change the state that other model defini-

tion rules depend on, causing them to add or remove objects (or tuples) to or from sets (or relations). These edges capture this dependence.

The following edges characterize how data structure updates may increase or decrease in the scope of quantifiers of consistency constraints or model definition rules. $\langle R_{ijklm}, F_w \rangle \in E$ (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 increase or decrease the scope of the model definition rule M_w.

7. **Edges from scope increase nodes to compensation nodes:** Compensation updates function to counteract increases in the scope of model definition rules; they are performed as a result of an increase in the scope of a model definition rule. These edges capture this dependence.

   $\langle S_w, \mathcal{R}_{wz} \rangle \in E$ 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 compensation updates that the repair algorithm may invoke to avoid inadvertently increasing the scope of a model definition rule.

8. **Edges from scope increase or decrease nodes to consequence nodes:** Changes in the scope of a model definition rule, if not compensated for, can affect the scope of other model definition rules or even violate consistency constraints. The consequence node is a placeholder that tracks these dependences. There is an edge from a scope node to the consequence node that tracks the dependences of changes in the scope of a model definition rule.

   $\langle S_w, \mathcal{C}_{wT} \rangle \in E$ and $\langle F_w, \mathcal{C}_{wF} \rangle \in E$ for each model definition rule M_w. These edges link the scope increase or decrease node to the consequences of increasing or decreasing the scope of a model definition rule.

9. **Edges from compensation update nodes to scope increase or decrease nodes:** Compensation updates change the concrete data structure. These changes can potentially make other model definition rules true or false. These edges capture this dependence.

$\langle \mathcal{R}_{wz}, F_{w'} \rangle \in E$ 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 $M_{w'}$.

10. **Edges from consequence nodes to scope increase or decrease nodes:** If a model definition rule changes scope, it causes objects (or tuples) to be added to or removed from sets (or relations). These changes to the model can cause additional changes in the scopes of model definition rules. These edges capture this dependence.

    $\langle \mathcal{C}_{wT}, F_{w'} \rangle \in E$ (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 other model definition rules.

11. **Edges from compensation update nodes to model repair nodes:** Compensation updates are designed to make a model definition rule remove an object (or tuple) from a set (or relation). Making a model definition rule remove an object (or tuple) may require making an inclusion constraint in the model definition rule true or false, or removing an object (or tuple) from a set (or relation) that the model definition rule quantifies over. As a result, a compensation update may need to perform a model repair. These edges capture this dependence.

    $\langle \mathcal{R}_{wz}, A_r \rangle \in E$ 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.

12. **Edges from consequence nodes to conjunction nodes:** If a model defini-

tion rule changes scope, it causes objects (or tuples) to be added to or removed from sets (or relations). These changes to the model can violate consistency constraints. These edges capture this dependence.

$\langle \mathcal{C}_{wT}, N_{ij} \rangle \in E$ 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}. These edges model the effects that increasing or decreasing the scope of a model definition rule may have on consistency constraints.

6.3 Interference

The edges in the repair dependence graph characterize either the potential effects of a repair action on the consistency constraints or what objects (or tuples) model definition rules add to sets (or relations). To place these edges, the graph construction algorithm must compute some basic interference relationships between repair actions and components of the consistency constraints and definition rules.

6.3.1 Model Repair Effects

There must be an edge from a model repair node to a conjunction node if the model repair may falsify the conjunction. The compiler uses the following procedure to determine if a model repair for a basic proposition may falsify a conjunction:

1. Our repair system desugars the domain and range sets of relations into consistency constraints as described in Section 4.4. In some cases, the repair tool is able to determine that model repairs that add tuples to relations always use objects from the appropriate domain or range set.

   If a conjunction corresponds to a constraint that ensures that a relation is well-formed (that objects in the tuples in the relation are in the appropriate domain or range set) and the model repair only adds tuples to the relation that satisfy this constraint, no edge is necessary.

2. If a model repair increases the size of a set (or relation) that the conjunction quantifies over, the constraint must hold on these new bindings. To ensure this, the repair algorithm may need to perform repairs to make a conjunction in the DNF form of the constraint true. The graph construction algorithm must place an edge from the model repair node to the conjunction node to represent this dependence.

3. Consider the pair of consistency constraint `for o in S, o.R=1 or o.S=2` and `for o in S, (!o.R=1) or o.S=3` where `S` is constrained to be a singleton set. If the repair algorithm performs an update that sets `o.S` equal to `2` to satisfy the first constraint, this implies that `o.R!=1`. Therefore, even though the model repair would normally potentially violate the second constraint, the analysis is able to determine that the model does not violate this constraint because `o.R!=1` implies that the constraint is true.

   In general, if the condition under which the model repair is performed ensures that the constraint containing the conjunction is true, even in the presence in the model repair, no edge is necessary.

4. If none of the previous rules apply, the specification compiler looks up the basic proposition being repaired and each of the basic propositions in the conjunction in the tables given in Figures 6-2, 6-3, 6-4, 6-5, 6-6, 6-7, 6-8, 6-9, 6-10, 6-11, 6-12, and 6-13. If a model repair may cause any of the basic propositions in the conjunction to be falsified, an edge must be placed from the model repair node to that conjunction node.

Figures 6-2, 6-3, 6-4, 6-5, 6-6, 6-7, 6-8, 6-9, 6-10, 6-11, 6-12, and 6-13 define an interferes predicate $\mathcal{IF}$ that captures the effect that a given model repair action may have on another basic proposition. This predicate returns false if the repair action does not falsify the basic proposition and true otherwise.

These figures were developed manually by reasoning about the proposition being repaired, the repair action, and the proposition that may be violated. The basic insights that were used to construct these tables follow:

- **Disjointness:** If a model repair only changes sets or relations that a basic proposition does not use, then that model repair cannot affect the basic proposition. This test appears in the figures as predicate of the form $S_1 = S_2$, $R_1 = R_2$, $\mathcal{U}(E, S)$,$\mathcal{U}(VE, S)$, $\mathcal{U}(E, R)$, or $\mathcal{U}(VE, R)$. The function $\mathcal{U}(E, S)$ evaluates to true if E depends on S.

- **Type of Action:** Some basic propositions can only be falsified by adding an object (or tuple) to a set(or relation), and others can only be falsified by removing an object (or tuple) from a set (or relation). For example, consider the basic proposition `size(S)<2`. Notice that this proposition cannot be falsified by removing objects from a set.

- **Reasoning about Inequalities:** Consider the effect of a model repair that adds objects to S to satisfy the constraint `size(S)=2` on the constraint `size(S)<5`. This model repair will not falsify the second basic proposition because it is only performed if the size of S is less than 2.

- **Reasoning about the Maximum Size of a Set:** Consider the effect of a model repair that adds objects to S to satisfy the constraint `x in S` on the constraint `size(S)<5`. If we can determine that the maximum size of S is less than 5, this model repair cannot falsify the second basic proposition because it is impossible to add more than 5 objects to S. Note that the function $\mathcal{MS}$ returns the maximum size of a set or relation.

- **Reasoning about Partitions:** Consider a model repair that adds a tuple to R to satisfy the constraint `for o1 in S1,size(o1.R)=3`. This model repair will not falsify a second basic proposition `for o2 in S2, size(o2.R)=1`, if the analysis can determine that the intersection of S1 and S2 is empty. We use partition information to determine this, and the function $\mathcal{NP}(\texttt{S1}, \texttt{S2})$ is true if and only if that partition constraints allow an object to be a member of both S1 and S2.

The definition of the interferes predicate $\mathcal{IF}$ uses several helper functions. The $\mathcal{MS}$ function returns the maximum size of the set or relation on which the function is evaluated. Section 6.3.4 gives the algorithm to compute this function. The function $\mathcal{U}$ takes two arguments, an expression E or VE and a set or relation, and returns true if the expression uses the given set or relation and false otherwise. The function $\mathcal{S}$ takes an expression VE or a variable V and returns a set that contains all of the objects that VE or V may reference. The function $\mathcal{NP}(S_1, S_2)$ is true if and only if the partition constraints allow an object to be a member of both S_1 and S_2. The function $\phi(E, V)$ is true if and only if E only uses on a single quantified variable V.

Addition to set S_1 to satisfy `size`(S_1)=c, `size`(S_1)`>=`c, `!size`(S_1)=$c-1$, or `!size`(S_1)`<=`$c-1$.

$\mathcal{IF}$(`size`(S_2)=c')	$(S_1 = S_2) \wedge (c' < c)$
$\mathcal{IF}$(`size`(S_2)`<=`c')	$(S_1 = S_2) \wedge (c' < c)$
$\mathcal{IF}$(`size`(S_2)`>=`c')	false
$\mathcal{IF}$(`!size`(S_2)=c')	$(S_1 = S_2) \wedge (c' = c)$
$\mathcal{IF}$(`!size`(S_2)`<=`c')	false
$\mathcal{IF}$(`!size`(S_2)`>=`c')	$(S_1 = S_2) \wedge (c' \leq c)$
$\mathcal{IF}$(V `in` S_2)	false
$\mathcal{IF}$(`!`V `in` S_2)	$S_1 = S_2$
$\mathcal{IF}$(VE *comp* E)	$\mathcal{U}(E, S_1)$

Addition to set S_1 to satisfy V `in` S_1.

$\mathcal{IF}$(`size`(S_2)=c')	$(S_1 = S_2) \wedge (\mathcal{MS}(S_2) > c')$
$\mathcal{IF}$(`size`(S_2)`<=`c')	$(S_1 = S_2) \wedge (\mathcal{MS}(S_2) > c')$
$\mathcal{IF}$(`size`(S_2)`>=`c')	false
$\mathcal{IF}$(`!size`(S_2)=c')	$(S_1 = S_2) \wedge (\mathcal{MS}(S_2) >= c')$
$\mathcal{IF}$(`!size`(S_2)`<=`c')	false
$\mathcal{IF}$(`!size`(S_2)`>=`c')	$(S_1 = S_2) \wedge (\mathcal{MS}(S_2) >= c')$
$\mathcal{IF}$(V `in` S_2)	false
$\mathcal{IF}$(`!`V `in` S_2)	$(S_1 = S_2)$
$\mathcal{IF}$(VE *comp* E)	$\mathcal{U}(E, S_1)$

Figure 6-2: Rules for computing interference from set additions

For model repair nodes that represent repairs that add or remove an object to a set, this procedure uses the rules given in Figure 6-2 or 6-3, respectively, to determine whether the model repair may falsify a conjunction. For model repair nodes that represent repairs that add or remove a tuple to a relation, this procedure uses the rules given in Figures 6-4, 6-5,6-6, and 6-7, or Figures 6-8, 6-9, 6-10, and 6-11, respectively, to determine whether the model repair may falsify a conjunction. For

Removal from set S_1 to satisfy `size`(S_1)=c, `size`(S_1)`<=`c, `!size`(S_1)=c+1, or `!size`(S_1)`>=`c+1.

$\mathcal{IF}$(`size`(S_2)=c')	$(S_1 = S_2) \wedge (c' > c)$
$\mathcal{IF}$(`size`(S_2)`<=`c')	false
$\mathcal{IF}$(`size`(S_2)`>=`c')	$(S_1 = S_2) \wedge (c' > c)$
$\mathcal{IF}$(`!size`(S_2)=c')	$(S_1 = S_2) \wedge (c' = c)$
$\mathcal{IF}$(`!size`(S_2)`<=`c')	$(S_1 = S_2) \wedge (c' \geq c)$
$\mathcal{IF}$(`!size`(S_2)`>=`c')	false
$\mathcal{IF}(V$ `in` $S_2)$	$S_1 = S_2$
$\mathcal{IF}($`!`V `in` $S_2)$	false
$\mathcal{IF}(VE$ *comp* $E)$	$\mathcal{U}(E, S_1)$

Removal from set S_1 to satisfy `!`V `in` S_1.

$\mathcal{IF}$(`size`(S_2)=c')	$(S_1 = S_2) \wedge (c' \neq 0)$
$\mathcal{IF}$(`size`(S_2)`<=`c')	false
$\mathcal{IF}$(`size`(S_2)`>=`c')	$S_1 = S_2$
$\mathcal{IF}$(`!size`(S_2)=c')	$S_1 = S_2$
$\mathcal{IF}$(`!size`(S_2)`<=`c')	$S_1 = S_2$
$\mathcal{IF}$(`!size`(S_2)`>=`c')	false
$\mathcal{IF}(V$ `in` $S_2)$	$S_1 = S_2$
$\mathcal{IF}($`!`V `in` $S_2)$	false
$\mathcal{IF}(VE$ *comp* $E)$	$\mathcal{U}(E, S_1)$

Figure 6-3: Rules for computing interference from set removals

model repair nodes that represent repairs that atomically modify a tuple in a relation, this procedure uses the rules given in Figures 6-12 and 6-13 to determine whether the model repair may falsify a conjunction.

If a repair changes whether a model definition rule adds an object (or tuple) to a set (or relation), this change to the model can result in further cascading changes. For example, a repair may cause one model definition rule to add a new object to a set, and a second model definition rule may quantify over this set and result in further changes to the model. There must be an edge from a model repair node to a scope increase or decrease node if the model repair may cause the scope of the model definition rule to increase or decrease. This can happen in one of two ways: the model repair may change the value of an inclusion constraint in the guard of the model definition rule or the model repair may change the scope of the model definition rule's quantifier. We use the same techniques that we used in the conjunction procedure to determine whether a model repair affects the scope of a model definition rule.

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 $\texttt{!size}(V_1.R_1)\texttt{<=}c-1$

$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' < c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge (c' < c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{size}(VE.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge (c' < c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(VE.R_2)\texttt{<=}c')$	$((R_1 = R_2) \wedge (c' < c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(VE.R_2)\texttt{>=}c')$	false
$\mathcal{IF}(V_3\ \texttt{in}\ V_2.R_2)$	false
$\mathcal{IF}(V_2\ \texttt{in}\ VE.R_2)$	false
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' = c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge (c' \leq c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(VE.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge (c' = c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(VE.R_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{!size}(VE.R_2)\texttt{>=}c')$	$((R_1 = R_2) \wedge (c' \leq c) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!}V_3\ \texttt{in}\ V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!}V_2\ \texttt{in}\ VE.R_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(V_2.R_2\ comp\ E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2)) \wedge (c > 1)) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(V_2.R'.R_2\ comp\ E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2.R')) \wedge (c > 1)) \vee \mathcal{U}(E, R_1)$ $\vee (R' = R_1)$
$\mathcal{IF}(VE.R'.R_2\ comp\ E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE.R')) \wedge (c > 1)) \vee \mathcal{U}(E, R_1) \vee$ $(R' = R_1) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(R_2.V_2\ comp\ E)$	$(R_1 = R_2) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(R_2.VE\ comp\ E)$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{size}(R_2.VE)\texttt{=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.VE)\texttt{<=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.VE)\texttt{>=}c')$	false
$\mathcal{IF}(V_3\ \texttt{in}\ R_2.V_2)$	false
$\mathcal{IF}(V_2\ \texttt{in}\ VE.V_2)$	false
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.VE)\texttt{=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.VE)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{!size}(R_2.VE)\texttt{>=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!}V_3\ \texttt{in}\ R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3))$
$\mathcal{IF}(\texttt{!}V_2\ \texttt{in}\ R_2.VE)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))) \vee \mathcal{U}(VE, R_1)$

Figure 6-4: Rules for computing interference from relation additions

Addition to a relation to satisfy `size`($VE_1.R_1$)=c, `size`($VE_1.R_1$)>=c, `!size`($VE_1.R_1$)=$c-1$, or `!size`($VE_1.R_1$)<=$c-1$

$\mathcal{IF}$(`size`($V_2.R_2$)=c')	$(R_1 = R_2) \wedge (c' < c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_2))$
$\mathcal{IF}$(`size`($V_2.R_2$)<=c')	$(R_1 = R_2) \wedge (c' < c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_2))$
$\mathcal{IF}$(`size`($V_2.R_2$)>=c')	false
$\mathcal{IF}$(`size`($VE_2.R_2$)=c')	$((R_1 = R_2) \wedge (c' < c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}$(`size`($VE_2.R_2$)<=c')	$((R_1 = R_2) \wedge (c' < c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}$(`size`($VE_2.R_2$)>=c')	false
$\mathcal{IF}$(V_3 `in` $V_2.R_2$)	false
$\mathcal{IF}$(V_2 `in` $VE_2.R_2$)	false
$\mathcal{IF}$(`!size`($V_2.R_2$)=c')	$(R_1 = R_2) \wedge (c' = c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_2))$
$\mathcal{IF}$(`!size`($V_2.R_2$)<=c')	false
$\mathcal{IF}$(`!size`($V_2.R_2$)>=c')	$(R_1 = R_2) \wedge (c' \leq c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_2))$
$\mathcal{IF}$(`!size`($VE_2.R_2$)=c')	$((R_1 = R_2) \wedge (c' = c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}$(`!size`($VE_2.R_2$)<=c')	false
$\mathcal{IF}$(`!size`($VE_2.R_2$)>=c')	$((R_1 = R_2) \wedge (c' \leq c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}$(`!`V_3 `in` $V_2.R_2$)	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_2))$
$\mathcal{IF}$(`!`V_2 `in` $VE_2.R_2$)	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}$($V_2.R_2$ *comp* E)	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_2)) \wedge (c > 1)) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}$($V_2.R'.R_2$ *comp* E)	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_2.R')) \wedge (c > 1)) \vee \mathcal{U}(E, R_1) \vee (R' = R_1)$
$\mathcal{IF}$($VE.R'.R_2$ *comp* E)	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE.R')) \wedge (c > 1)) \vee \mathcal{U}(E, R_1) \vee (R' = R_1) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}$($R_2.V_2$ *comp* E)	$(R_1 = R_2) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}$($R_2.VE$ *comp* E)	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}$(`size`($R_2.V_2$)=c')	$R_1 = R_2$
$\mathcal{IF}$(`size`($R_2.V_2$)<=c')	$R_1 = R_2$
$\mathcal{IF}$(`size`($R_2.V_2$)>=c')	false
$\mathcal{IF}$(`size`($R_2.VE_2$)=c')	$(R_1 = R_2) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}$(`size`($R_2.VE_2$)<=c')	$(R_1 = R_2) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}$(`size`($R_2.VE_2$)>=c')	false
$\mathcal{IF}$(V_3 `in` $R_2.V_2$)	false
$\mathcal{IF}$(V_2 `in` $VE_2.V_2$)	false
$\mathcal{IF}$(`!size`($R_2.V_2$)=c')	$R_1 = R_2$
$\mathcal{IF}$(`!size`($R_2.V_2$)<=c')	false
$\mathcal{IF}$(`!size`($R_2.V_2$)>=c')	$R_1 = R_2$
$\mathcal{IF}$(`!size`($R_2.VE_2$)=c')	$(R_1 = R_2) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}$(`!size`($R_2.VE_2$)<=c')	false
$\mathcal{IF}$(`!size`($R_2.VE_2$)>=c')	$(R_1 = R_2) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}$(`!`V_3 `in` $R_2.V_2$)	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_3))$
$\mathcal{IF}$(`!`V_2 `in` $R_2.VE_2$)	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_2))) \vee \mathcal{U}(VE_2, R_1)$

Figure 6-5: Rules for computing interference from relation additions

Addition to relation to satisfy V' `in` $V_1.R_1$

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

Figure 6-6: Rules for computing interference from relation additions

Addition to relation to satisfy V' `in` $VE_1.R_1$

$\mathcal{IF}(\texttt{size}(V.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V)) \wedge (\mathcal{MS}(R_2) > c')$
$\mathcal{IF}(\texttt{size}(V.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V)) \wedge (\mathcal{MS}(R_2) \geq c')$
$\mathcal{IF}(\texttt{size}(V.R_2)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{size}(VE_2.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2)) \wedge (\mathcal{MS}(R_2) > c')) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{size}(VE_2.R_2)\texttt{<=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2)) \wedge (\mathcal{MS}(R_2) \geq c')) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{size}(VE_2.R_2)\texttt{>=}c')$	false
$\mathcal{IF}(V_3 \texttt{ in } V.R_2)$	false
$\mathcal{IF}(V_3 \texttt{ in } VE_2.R_2)$	false
$\mathcal{IF}(\texttt{!size}(V.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{!size}(V.R_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{!size}(V.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V)) \wedge (\mathcal{MS}(R_2) \geq c')$
$\mathcal{IF}(\texttt{!size}(VE_2.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(VE_2.R_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{!size}(VE_2.R_2)\texttt{>=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2)) \wedge (\mathcal{MS}(R_2) \geq c')) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } V.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V)) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_3)$
$\mathcal{IF}(\texttt{!}V \texttt{ in } VE_2.R_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V)) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V)) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(V.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(V.R'.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V.R'))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(VE.R'.R_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE.R'))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(R_2.V_2 \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(R_2.VE \textit{ comp } E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))) \vee \mathcal{U}(VE, R_1) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V)) \wedge (\mathcal{MS}(R_2) > c')$
$\mathcal{IF}(\texttt{size}(R_2.V)\texttt{<=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V)) \wedge (\mathcal{MS}(R_2) > c')$
$\mathcal{IF}(\texttt{size}(R_2.V)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{size}(R_2.VE_2)\texttt{=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(VE_2)) \wedge (\mathcal{MS}(R_2) > c')) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.VE_2)\texttt{<=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(VE_2)) \wedge (\mathcal{MS}(R_2) > c')) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.VE_2)\texttt{>=}c')$	false
$\mathcal{IF}(V_3 \texttt{ in } R_2.V)$	false
$\mathcal{IF}(V \texttt{ in } VE_2.V)$	false
$\mathcal{IF}(\texttt{!size}(R_2.V)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V)) \wedge (\mathcal{MS}(R_2) \geq c')$
$\mathcal{IF}(\texttt{!size}(R_2.V)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{!size}(R_2.V)\texttt{>=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V)) \wedge (\mathcal{MS}(R_2) \geq c')$
$\mathcal{IF}(\texttt{!size}(R_2.VE_2)\texttt{=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(VE_2)) \wedge (\mathcal{MS}(R_2) \geq c')) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.VE_2)\texttt{<=}c')$	false
$(\mathcal{IF}(\texttt{!size}(R_2.VE_2)\texttt{>=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(VE_2)) \wedge (\mathcal{MS}(R_2) \geq c')) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!}V_3 \texttt{ in } R_2.V)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_3) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{!}V \texttt{ in } R_2.VE_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$

Figure 6-7: Rules for computing interference from relation additions

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

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

Figure 6-8: Rules for computing interference from relation removals

Removal from a relation to satisfy $\texttt{size}(VE_1.R_1)\texttt{=}c$, $\texttt{size}(VE_1.R_1)\texttt{<=}c$, $\texttt{!size}(VE_1.R_1)\texttt{=}c+1$, or $\texttt{!size}(VE_1.R_1)\texttt{>=}c+1$

$\mathcal{IF}(\texttt{size}(V.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' > c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{size}(V.R_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{size}(V.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge (c' > c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{size}(VE_2.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge (c' > c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{size}(VE_2.R_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{size}(VE_2.R_2)\texttt{>=}c')$	$((R_1 = R_2) \wedge (c' > c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(V_3\ \texttt{in}\ V.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(V\ \texttt{in}\ VE_2.R_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(V.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' = c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{!size}(V.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge (c' \geq c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{!size}(V.R_2)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{!size}(VE_2.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge (c' = c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(VE_2.R_2)\texttt{<=}c')$	$((R_1 = R_2) \wedge (c' \geq c) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(VE_2.R_2)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{!}V_2\ \texttt{in}\ V.R_2)$	false
$\mathcal{IF}(\texttt{!}V\ \texttt{in}\ \ vexpr.R_2)$	false
$\mathcal{IF}(V.R_2\ comp\ E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(V.R'.R_2\ comp\ E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V.R'))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(VE.R'.R_2\ comp\ E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE.R'))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(R_2.V\ comp\ E)$	$(R_1 = R_2) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(R_2.VE_2\ comp\ E)$	$(R_1 = R_2) \vee \mathcal{U}(VE_2, R_1) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{size}(R_2.V)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.VE_2)\texttt{=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.VE_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{size}(R_2.VE_2)\texttt{>=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(V_3\ \texttt{in}\ R_2.V)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_3))$
$\mathcal{IF}(V\ \texttt{in}\ R_2.VE_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.V)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{!size}(R_2.VE_2)\texttt{=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.VE_2)\texttt{<=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.VE_2)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{!}V_3\ \texttt{in}\ R_2.V)$	false
$\mathcal{IF}(\texttt{!}V\ \texttt{in}\ R_2.VE_2)$	false

Figure 6-9: Rules for computing interference from relation removals

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

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

Figure 6-10: Rules for computing interference from relation removals

Removal from a relation to satisfy $\texttt{!}V'$ $\texttt{in}$ $VE_1.R_1$.

$\mathcal{IF}(\texttt{size}(V.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{size}(V.R_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{size}(V.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{size}(VE_2.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{size}(VE_2.R_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{size}(VE_2.R_2)\texttt{>=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(V_2$ $\texttt{in}$ $V.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(V$ $\texttt{in}$ $VE_2.R_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(V.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{!size}(V.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{!size}(V.R_2)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{!size}(VE_2.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(VE_2.R_2)\texttt{<=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(VE_2.R_2)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{!}V_2$ $\texttt{in}$ $V.R_2)$	false
$\mathcal{IF}(\texttt{!}V$ $\texttt{in}$ $VE_2.R_2)$	false
$\mathcal{IF}(V.R_2$ *comp* $E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(V.R'.R_2$ *comp* $E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V.R'))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(VE.R'.R_2$ *comp* $E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(VE.R'))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(R_2.V_2$ *comp* $E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(R_2.VE$ *comp* $E)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V_2))) \vee \mathcal{U}(VE, R_1) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{size}(R_2.V)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{size}(R_2.V)\texttt{>=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{size}(R_2.VE_2)\texttt{=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.VE_2)\texttt{<=}c')$	false
$\mathcal{IF}(\texttt{size}(R_2.VE_2)\texttt{>=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(V_2$ $\texttt{in}$ $R_2.V)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V_2))$
$\mathcal{IF}(V$ $\texttt{in}$ $R_2.VE_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(VE_1), \mathcal{S}(V))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.V)\texttt{=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{!size}(R_2.V)\texttt{<=}c')$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(V))$
$\mathcal{IF}(\texttt{!size}(R_2.V)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{!size}(R_2.VE_2)\texttt{=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.VE_2)\texttt{<=}c')$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V'), \mathcal{S}(VE_2))) \vee \mathcal{U}(VE_2, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.VE_2)\texttt{>=}c')$	false
$\mathcal{IF}(\texttt{!}V_2$ $\texttt{in}$ $R_2.V)$	false
$\mathcal{IF}(\texttt{!}V$ $\texttt{in}$ $R_2.VE_2)$	false

Figure 6-11: Rules for computing interference from relation removals

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

$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' \neq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge (c' = 0) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge (c' > 1) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(VE.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge (c' \neq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(VE.R_2)\texttt{<=}c')$	$((R_1 = R_2) \wedge (c' = 0) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(VE.R_2)\texttt{>=}c')$	$((R_1 = R_2) \wedge (c' > 1) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(V_3\ \texttt{in}\ V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(V_2\ \texttt{in}\ VE.R_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' = 1) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge (c' \geq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge (c' \leq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(VE.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge (c' = 1) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(VE.R_2)\texttt{<=}c')$	$((R_1 = R_2) \wedge (c' \geq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(VE.R_2)\texttt{>=}c')$	$((R_1 = R_2) \wedge (c' \leq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!}V_3\ \texttt{in}\ V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!}V_2\ \texttt{in}\ VE.R_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(V_2.R_2\ comp_2\ E_2)$	$(((comp_1 \neq comp_2) \vee (E_1 \neq E_2[V_2/V_1]) \vee \phi(E_1, V_1) \vee \phi(E_2, V_2)) \wedge$ $(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2))) \vee \mathcal{U}(E_2, R_1)$
$\mathcal{IF}(V_2.R'.R_2\ comp_2\ E_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_2.R'))) \vee \mathcal{U}(E_2, R_1) \vee (R_1 = R')$
$\mathcal{IF}(VE.R'.R_2\ comp_2\ E_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(VE.R'))) \vee \mathcal{U}(E_2, R_1) \vee (R_1 = R') \vee$ $\mathcal{U}(VE, R_1)$
$\mathcal{IF}(R_2.V_2\ comp\ E)$	$(R_1 = R_2) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(R_2.VE\ comp\ E)$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.VE)\texttt{=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.VE)\texttt{<=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.VE)\texttt{>=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(V_3\ \texttt{in}\ R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3))$
$\mathcal{IF}(V_3\ \texttt{in}\ R_2.VE)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.VE)\texttt{=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.VE)\texttt{<=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.VE)\texttt{>=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!}V_3\ \texttt{in}\ R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3))$
$\mathcal{IF}(\texttt{!}V_3\ \texttt{in}\ R_2.VE)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1), \mathcal{S}(V_3))) \vee \mathcal{U}(VE, R_1)$

Figure 6-12: Rules for computing interference from relation modifications

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

$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' \neq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge (c' = 0) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge (c' > 1) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{size}(VE.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge (c' \neq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(VE))) \vee$ $\mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(VE.R_2)\texttt{<=}c')$	$((R_1 = R_2) \wedge (c' = 0) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(VE))) \vee$ $\mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(VE.R_2)\texttt{>=}c')$	$((R_1 = R_2) \wedge (c' > 1) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(VE))) \vee$ $\mathcal{U}(VE, R_1)$
$\mathcal{IF}(V_3\ \texttt{in}\ V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{=}c')$	$(R_1 = R_2) \wedge (c' = 1) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{<=}c')$	$(R_1 = R_2) \wedge (c' \geq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(V_2.R_2)\texttt{>=}c')$	$(R_1 = R_2) \wedge (c' \leq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!size}(VE.R_2)\texttt{=}c')$	$((R_1 = R_2) \wedge (c' = 1) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(VE))) \vee$ $\mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(VE.R_2)\texttt{<=}c')$	$((R_1 = R_2) \wedge (c' \geq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(VE))) \vee$ $\mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(VE.R_2)\texttt{>=}c')$	$((R_1 = R_2) \wedge (c' \leq 1) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(VE))) \vee$ $\mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!}V_3\ \texttt{in}\ V_2.R_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))$
$\mathcal{IF}(\texttt{!}V_2\ \texttt{in}\ VE.R_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(VE))) \vee \mathcal{U}(VE, R_2)$
$\mathcal{IF}(V_2.R_2\ comp_2\ E_2)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))) \vee \mathcal{U}(E_2, R_1)$
$\mathcal{IF}(V_2.R''_1...R''_{n'}.R_2\ comp_2\ E_2)$	$(((comp_1 \neq comp_2) \vee (E_1 \neq E_2[V_2/V_1]) \vee \phi(E_1, V_1) \vee \phi(E_2, V_2) \vee$ $(n' \neq n) \vee (R'_1 \neq R''_1) \vee ... \vee (R'_n \neq R''_n)) \wedge (R_1 = R_2) \wedge$ $\mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2.R''_1...R''_{n'}))) \vee \mathcal{U}(E_2, R_1) \vee (R''_1 = R_1) \vee$ $... \vee (R''_{n'} = R_1)$
$\mathcal{IF}(R_2.V_2\ comp\ E)$	$(R_1 = R_2) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(R_2.VE\ comp\ E)$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1) \vee \mathcal{U}(E, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{size}(R_2.VE)\texttt{=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.VE)\texttt{<=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{size}(R_2.VE)\texttt{>=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(V_3\ \texttt{in}\ R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_3))$
$\mathcal{IF}(V_2\ \texttt{in}\ R_2.VE)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{<=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.V_2)\texttt{>=}c')$	$R_1 = R_2$
$\mathcal{IF}(\texttt{!size}(R_2.VE)\texttt{=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.VE)\texttt{<=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!size}(R_2.VE)\texttt{>=}c')$	$(R_1 = R_2) \vee \mathcal{U}(VE, R_1)$
$\mathcal{IF}(\texttt{!}V_3\ \texttt{in}\ R_2.V_2)$	$(R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_3))$
$\mathcal{IF}(\texttt{!}V_2\ \texttt{in}\ R_2.VE)$	$((R_1 = R_2) \wedge \mathcal{NP}(\mathcal{S}(V_1.R'_1...R'_n), \mathcal{S}(V_2))) \vee \mathcal{U}(VE, R_1)$

Figure 6-13: Rules for computing interference from relation modifications

6.3.2 Data Structure and Compensation Updates

Performing a data structure update modifies the concrete data structure. In addition to the model repair that the update is designed to implement, the update may cause additional unintended increases or decreases in the scopes of the model definition rules. The edges from the model repair node already capture the dependences resulting from the intended model repair. We use edges from the data structure or compensation update node to capture these additional effects that a data structure or compensation update may have on the scopes of other model definition rules.

The compiler uses a rule-based procedure to determine whether a data structure update has any additional effects. The default rule is that updating a field `f` in the concrete data structures may either increase or decrease the scope of any model definition rule that uses `f`, requiring an insertion of an edge from the update node to the scope increase or decrease node for the model definition rule in the repair dependence graph. The algorithm implements exceptions to this rule (and omits the corresponding edges in these cases) for the intended effect of the data structure update, initial additions to a set, updates that affect only a single binding of a model definition rule, and recursive data structures.

- **Initial addition:** The specification compiler determines that an update adds the first element to a previously empty set, then the update does not decrease the scope of any model definition rule that may add an object to that set.

  A similar rule applies to relations. Consider, for example, a model definition rule with an inclusion constraint of the form $\langle V, FE \rangle$ and an update that adds the tuple $\langle o_1, o_2 \rangle$. If the specification compiler determines that an update adds the first object to the image of a given object under the relation, and the updated fields or array elements can be determined to be used only by the quantifier binding where $V = o_1$, then the update does not cause any scope decreases for that model definition rule.

- **Updates to newly allocated objects:** An update that only writes to a newly allocated object does not decrease the scope of any model definition rule.

- **Self Propagation:** Consider an update intended to add an object (or tuple) to a set (or relation), or an update that directly modifies the value of a tuple in a relation. If the state updated by the data structure update is only referenced by a single quantifier binding of the model definition rule used to construct the update, then the update does not further increase the scope of that model definition rule. Similar reasoning holds for the cause of decreasing the scope of a model definition rule.

  Consider an update that performs an atomic modify operation. If the state updated by the data structure update is only referenced by a single quantifier binding of the model definition rule used to construct the update, then the update does not further increase or decrease the scope of that model definition rule.

- **Recursive data structures:** Consider an update that adds or removes an object to a recursive data structure. For such a recursive data structure, the model definition rules typically contain a base case rule of the form `for` V' `in` $S', G \Rightarrow FE$ `in` S and a recursive case rule of the form `for` V `in` $S, G' \Rightarrow FE'$ `in` S.

  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' that 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 that 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` $= V$ **does not falsify** $!V$.`field` $= NULL$**:** If an update satisfies the proposition V.`field` $= V$ where V is a quantification variable, then the update

does not increase the scope of any model definition rule through the proposition $V.\texttt{field} = NULL$. The update also does not decrease the scope of any model definition rule that contains the proposition $!V.\texttt{field} = NULL$.

- $V.\texttt{field} = NULL$ **does not falsify** $!V.\texttt{field} = V$**:** If an update satisfies the proposition $V.\texttt{field} = NULL$, then the update does not increase the scope of any model definition rule through the proposition $V.\texttt{field} = V'$ and does not decrease the scope of any model definition rule through the proposition $!V.\texttt{field} = V'$.

- $V.\texttt{field} = E$ **does not falsify** $V.\texttt{field} = E$ **and does not satisfy** $!V.\texttt{field} = E$**:** If an update satisfies the proposition $V.\texttt{field} = E$ where E depends solely on V and globals, then the update does not increase the scope of any model definition rule through the proposition $!V.\texttt{field} = E$ and does not decrease the scope of any model definition rule through the proposition $V.\texttt{field} = E$.

- $FE.\texttt{field} = E$ **does not falsify** $FE.\texttt{field} = E$ **and does not satisfy** $!FE.\texttt{field} = E$**:** If an update satisfies the proposition $FE.\texttt{field} = E$ where E depends solely on V and globals, and FE does not contain any pointer dereferences, then the update does not increase the scope of any model definition rule through the proposition $!FE.\texttt{field} = E$ and does not decrease the scope of any model definition rule through the proposition $FE.\texttt{field} = E$.

- $V.\texttt{field}[E_1] = E_2$ **does not falsify** $V.\texttt{field}[E_1] = E_2$ **and does not satisfy** $!V.\texttt{field}[E_1] = E_2$**:** If an update satisfies the proposition $V.\texttt{field}[E_1] = E_2$ where E_1 and E_2 depend solely on V and globals, and E_1 does not contain any dereferences, then the update does not increase the scope of any model definition rule through the proposition $!V.\texttt{field}[E_1] = E_2$ and does not decrease the scope of any model definition rule through the proposition $V.\texttt{field}[E_1] = E_2$.

- $FE.\texttt{field}[E_1] = E_2$ **does not falsify** $FE.\texttt{field}[E_1] = E_2$ **and does not satisfy** $!FE.\texttt{field}[E_1] = E_2$**:** If an update satisfies the proposition $FE.\texttt{field}[E_1] = E_2$ where E_1 and E_2 depend solely on V and globals, E_1 does

not contain any dereferences, and FE does not contain any pointer dereferences, then the update does not increase the scope of any model definition rule through the proposition $!FE.\texttt{field}[E_1] = E_2$ and does not decrease the scope of any model definition rule through the proposition $FE.\texttt{field}[E_1] = E_2$.

6.3.3 Scope Increases and Decreases

Increases or decreases in the scope of a model definition rule 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 consistency constraints that depend on the set (or relation) or cause additional changes in the scopes of other model definition rules. The repair dependence graph contains edges that account for these possibilities.

The following rules determine whether a change in scope represented by one scope increase or decrease node may cause a change in scope represented by a 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 that tests membership in the same set (or relation). Increases in the scope of a model definition rule that constructs a set (or relation) may decrease the scope of any model definition rule that includes a negated guard that tests 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 that tests 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 that tests membership in the same set (or relation).

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

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

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

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

Figure 6-14: Rules for computing interference from model definition rule scope changes

6.3.4 Computing Upper Bounds on the Sizes of Sets and Relations

The algorithm for constructing the repair dependence graph can use upper bounds on the sizes of sets to improve the precision of the repair dependence graph. For example, consider the constraint `for a in A, for b in B, b.R=a.R'`, where `A` has at most one element by construction. If the specification compiler is able to determine that the maximum size of `A` is 1, it can show that repairs of the relation `R` do not falsify other bindings of the same constraint. The specification compiler contains an analysis to compute conservative upper bounds on the sizes of the sets and the relations.

The analysis begins by assuming that all sets and relations are empty. Then for each set (or relation), the analysis iterates through all of the model definition rules that may add objects (or tuples) to the respective set (or relation). The analysis looks up the sizes of the sets, relations, or ranges that the variables in the model definition rule quantify over. The analysis then computes that the size of the set (or relation)

that the model definition rules adds objects (or tuples) to is conservatively bound by the number of variable bindings of all the model definition rules that may add objects (or tuples) to that set (or relation). To guarantee termination of this algorithm, if the computed size of a set or a relation exceeds a fixed bound, the analysis conservatively assumes that this set or relation may be unbounded in size. Without this bound mechanism, the analysis would run forever on model definition rules that traverse unbounded linked data structures. The algorithm continues this computation until it reaches a fixed point.

Sometimes it is useful to define a set that contains the object at the end of a linked list. We would like the size analysis to be able to compute accurate bounds for such sets. To do this the analysis looks for sets (or relations) that are constructed by a set of model definition rules that add a finite number of objects (or tuples) to a set (or relation) and a single recursively defined model definition rule that quantifies over that set (or relation) and potentially adds a single object (or tuple) to that set (or relation) for each object (or tuple) in that set (or relation). Then the analysis looks for other model definition rules that quantify over the same set (or relation). If the guards of these model definition rules are mutually exclusive with the recursively defined model definition rule, then the number of bindings that these model definition rules can have for that set (or relation) is equal to the number of objects (or tuples) added to the set (or relation) by the non-recursively defined model definition rules. Intuitively, the analysis exploits the fact that the number of ends for linked lists is at most equal to the number of beginnings for linked lists. Determining that a set which contains the last object in a linked list has at most one element allows our system to repair pointers to the end of linked lists.

6.4 Behavior of the Generated Repair Algorithms

The repair dependence graph captures all possible behaviors of the generated repair algorithm. The developer can discover all of the possible ways that the repair algorithm may satisfy a constraint by examining the corresponding conjunction nodes. In

particular, each conjunction node represents a conjunction in the disjunctive normal form of a constraint that the repair algorithm may satisfy to repair a violation of that constraint. The model repair nodes represent model repairs (such as adding an object to a set) that the repair algorithm may perform to satisfy the individual basic propositions making up a conjunction. By examining these nodes, the developer can discover what model repairs the repair algorithm may perform. Finally, the update nodes tell the developer how the repair algorithm may implement a model repair. For each update node, the specification compiler outputs a corresponding description that lists the write operations (both the field or array element to be written to and the value written) that the repair algorithm performs. Furthermore, the edges capture any potential effects of the repair actions.

6.5 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 conjunction node captures all of the possible effects of repairing that model conjunction. Intuitively, any possible infinite repair sequence must therefore show up as a cycle.

If the repair dependence graph is 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 then the corresponding repair algorithm terminates. [1] The specification compiler may prune model conjunction nodes, data structure update nodes, consequence nodes, and compensation update nodes to satisfy these cyclity constraints. The generated repair algorithm never performs the repair actions that correspond to deleted nodes. The final graph must satisfy the following conditions in order to ensure that a repair exists for any violated constraint:

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

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

2. Each model 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.

Theorem: If the repair dependence 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.

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

6.5.1 Individual Model Repairs Terminate

Since a model repair action is always completed before the next model repair action starts, the repair algorithm can only perform repair actions that are reachable in the repair graph from the model repair node without traversing a model conjunction node.

This reachable subgraph contains the following classes of nodes: model repair nodes, data structure update nodes, scope increase or decrease 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 model repair 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 repair algorithm performs the initial data structure updates (which trivially terminate), it only performs operations for the compensation update nodes.

It is clear that individual model repair actions terminate. Consider the strongly connected components of the subgraph in topologically sorted order. Note that while it is possible that a strongly connected component to contain more than one node, the restrictions on cycles ensure that such sets of nodes do not represent any actual work. Once the first compensation update node is used to prevent all of the initial undesired increases 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 compensation update node has been finished, no further activations of the next rule will occur, and so forth.

Since individual model repairs terminate, any infinite repair chains must involve infinitely many model repairs.

6.5.2 Repair Terminates

Now that we have shown that individual model 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.

Correctness Argument: (Structural induction).

(Base Case:) The base case, an acyclic graph with '0' nodes, 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. As a result, the repair algorithm will not add objects (or tuples) to the sets (or relations) that the conjunction quantifies over. 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. Therefore, 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 terminate.

6.6 Pruning the Repair Dependence Graph

Conjunction nodes, model repair nodes, data structure update nodes, or compensation update nodes correspond to actions that the repair algorithm must perform. Therefore, cycles that are reachable and contain these nodes correspond to a potential non-terminating computation. If the repair dependence graph contains such a cycle, the specification compiler searches for a set of conjunction nodes, update nodes, compensation nodes, and consequence nodes that, if removed from the graph, remove these cycles from the graph. Unfortunately, this search space is exponential in the number of nodes to be searched. We have developed a set of rules to reduce this search space. The specification compiler uses these rules to determine that a node must remain graph, that a node must be removed from the graph, or that a node can always safely be left in the graph. In each case, the specification compiler can remove the given node from the set of nodes it searches. After these rules have been

evaluated, the specification compiler performs a search on the remaining nodes. The specification compiler uses the following sets of rules:

Rules to determine that a node must remain in the graph:

1. The repair algorithm must have at least one repair option for each constraint. Since conjunction nodes correspond to repair options for a constraint, a conjunction node that represents the only way to satisfy a constraint must remain in the graph.

2. If a conjunction node must remain in the graph, the repair algorithm has to be able to perform all of the corresponding model repairs. To perform a model repair, the repair algorithm must have at least one data structure update that implements the given model repair. If a data structure update node represents the only implementation of a model repair for a conjunction node that must remain in the graph, then that data structure update node must remain in the graph.

3. If the scope of a model definition rule changes, the repair algorithm can choose to either perform a compensation update or to allow the scope change to occur. The compensation update nodes and consequence nodes represent these two choices. If a given scope node references only one consequence or compensation update node, then the choice is trivial and that compensation or consequence node must remain in the graph.

Rules to determinate that a node must be removed from the graph:

1. This rule examines a data structure update, the corresponding model repair node, and the corresponding conjunction node (if any) in the context of the nodes that must remain in the graph. If this set of nodes contains a cycle through the data structure update node, then if the data structure update node remains in the final graph it will also contain that cycle. Therefore, the data structure update node must be removed.

2. This rule examines a scope node and its corresponding compensation node in the context of the nodes that must remain in the graph. If this set of nodes contains a cycle through the compensation node, then if the compensation update node remains in the final graph it will also contain that cycle. Therefore, the compensation update node must be removed.

3. Model repairs that modify a relation must be implemented with either a data structure update that modifies the relation or a pair of data structure updates in which one update removes a tuple from the relation and the other update adds a tuple to the relation. If one data structure update in this pair is not present, the other data structure update is not useful by itself and must be removed from the graph.

Rule to determine that a node can safely remain in the graph

1. If a consequence node, a data structure update node, a compensation update node, or a conjunction node in the context of all nodes that could be in the graph has 1) no cycles through it, 2) does not result in a cycle becoming reachable through it, and 3) the model repair and scope nodes that become reachable as a result of the addition of this node have sufficient compensation, consequence, and update nodes, then this node will not have any cycles through it in the complete graph and has all of the necessary actions to be implemented. Therefore, it is always safe to place this node in the graph.

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

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

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

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

Figure 6-15: Rules for computing interference from model definition rule scope changes

Chapter 7

Bounds on the Repair Overhead

The performance of the automatically-generated repair algorithm depends on the specification. Specifications that contain tightly-coupled constraints, i.e. systems of constraints in which the repair of one constraint may invalidate a second, may require more repair actions than would otherwise be expected due to possible cascading behavior. The developer can use the repair dependence graph to reason about the worst-case performance of the automatically generated repair algorithm.

7.1 Bound on the Number of Repair Actions

The repair dependence graph provides a means to calculate an upper bound on the number of repair actions performed by the repair process in terms of the initial size of the data structure. We present an algorithm that, for an acyclic repair dependence graph, computes an upper bound on the number of repairs performed. [1] Since the graph is acyclic, we simply analyze it in topological order. For each node, the algorithm computes an upper bound on the number of repair actions that correspond to the node and the number of objects (or tuples) that the action corresponding to the node may cause to be added to a set (or relation). To calculate the number of

[1] The termination analysis ensures that cycles do not contain model repair nodes, data structure update nodes, compensation update nodes, or conjunction nodes. Analyzing the remaining cycles requires additional information about the implementation of object allocators to bound the number of times the cycle can be performed.

bindings for a given node, we calculate the maximum sizes of the sets or relations the node quantifies over (by summing the initial size of the set or relation and the number of objects or tuples that the nodes along the incoming edges may add). We then multiply the sizes of all the sets or relations that are quantified over to get the total number of bindings for the node.

- **Conjunction Nodes N_{ij}:** The number of times that the repair algorithm may repair a conjunction node is bounded by one more than the sum of the repairs of the nodes that reference this node multiplied by the number of bindings for that conjunction node.

- **Model Repair Nodes A_{ijkl}/A_r:** The number of times the repair algorithm may perform a model repair is bounded by the sum of the number of repair actions of the nodes that reference this node. If the model repair adds an object or tuple to a set or relation, the number of additions to the set or relation that it performs is equal to its number of repairs multiplied by the maximum number of additions it may perform to satisfy a single instance of a predicate. For example, a model repair action corresponding to the constraint `size(S)=6` may add 6 objects to the set `S`.

- **Data Structure Update Nodes R_{ijklm}/R_{rs}:** The number of times the repair algorithm may perform a data structure update is bounded by the sum of the number of repairs of the nodes that reference this node.

- **Scope Increase Nodes S_w:** The number of repairs a scope increase node may cause is bounded by the sum of the repairs of the nodes that reference this node. The number of objects or tuples that may be added to a set or relation by increases in the scope of the model definition rule is bounded by the sum of the repairs of the nodes that reference this node multiplied by the number of bindings for that model definition rule.

- **Scope Decrease Nodes F_w:** The number of repairs a scope decrease node may cause is bounded by the sum of the repairs of the nodes that reference this

node.

- **Compensation Update Nodes $\mathcal{R}_{wz}$:** The number of times the repair algorithm may perform a compensation update is bounded by the number of repairs performed by the scope increase node that references it multiplied by the number of bindings for that scope increase node.

- **Consequence Nodes $\mathcal{C}_{wT}/\mathcal{C}_{wF}$:** The number of repair actions for a consequence node is equal to the number of repair actions for the corresponding scope increase or decrease node. The number of additions is equal to the number of additions for the corresponding scope increase node.

This algorithm provides a method to calculate an upper bound on the number of repairs performed by the repair algorithm. The number of times the repair algorithm rebuilds the model is bounded by the sum of the number of conjunctions repaired, the number of repairs of compensation nodes, the number of model repairs that modify a relation, and the number of model repairs that remove an object or tuple from a set or relation. The number of times the model is checked is bounded by the sum of the conjunctions repaired.

Note that the upper bound is likely to be very conservative for most systems of constraints. For example, typically the repair of one constraint will result in the violation of only one binding of another constraint. The developer can adapt the basic technique to take advantage of additional information.

7.2 Reducing the Repair Overhead

For extremely large data structures with complex specifications, the overhead of performing repairs may become significant. There are two ways of reducing this overhead: reducing the number of individual repairs that the repair algorithm performs and lowering the cost of each individual repair. Both of these approaches may improve the worst-case performance of the repair algorithm.

7.2.1 Reducing the Number of Repairs

The number of repairs performed by the repair algorithm to satisfy a specification can depend on the repair actions that the repair algorithm chooses to perform. In some cases, the repair algorithm can make a trade off between the quality of the repair and the number of repair actions.

Our current implementation uses a cost function to select conjunctions to repair. We intended for this cost function to optimize the quality of the repairs selected. In some cases, for performance reasons, it may be desirable to minimize the number of additional repair actions an initial repair may trigger. The repair analysis can use the repair dependence graph to calculate an upper bound on the number of additional repair actions that a given repair choice may trigger. These upper bounds can be used to select the repair actions that minimize the upper time bound of the repair algorithm.

7.2.2 Reducing the Model Building Overhead

The current implementation completely rebuilds the model after each model repair. If satisfying a specification involves many repair actions, this overhead can become significant. The cost of rebuilding the model can be amortized over many repair actions and/or only the affected portions of the model can be rebuilt.

Amortizing the Model Rebuilding Costs The current implementation rebuilds the abstract model after each conjunction repair. The implementation could perform many model repairs before rebuilding the model to amortize the model rebuilding overhead. For example, for the file system example in Section 3 the model is rebuilt every time a bit in the BlockBitmap is updated. A more efficient repair algorithm would rebuild the model once for all of the updates to bits in the BlockBitmap.

This change would significantly reduce the time taken to perform a repair for repairs that involve many data structure updates. However, this may not translate into any significant real world performance gains, because we expect repairs to be

performed relatively infrequently for most applications.

Partial Model Rebuilding Many repairs only change a portion of the abstract model. The repair dependence graph can be used to determine what portion of the abstract model a given repair may invalidate. This information can be used to partially reconstruct the abstract model between repair actions.

7.3 Reducing the Checking Overhead

For small data structures, the overhead of performing frequent consistency checks is unlikely to be significant. However, for large data structures or systems which need to perform very frequent consistency checking, this overhead can become significant. We propose three methods for reducing the checking overhead.

7.3.1 Check On Failure

The system can augment the application to monitor for faults such as divide by zero and segmentation fault violations. Because inconsistent data structures often cause such faults, the signal handler invokes the check and repair algorithm, and then resumes the execution at the nearest safe point.

This mechanism for triggering the repair algorithm avoids the overheads associated with performing consistency checks during the normal execution of the application. Unfortunately, the system will fail to perform repairs if an inconsistency is not externally detectable.

7.3.2 Partial Specification Checking

It is often the case that a method in an application is concerned with only a portion of the objects in the application or a portion of the consistency constraints. If the consistency constraints that involve these objects are enforced then from the method's view the data structure is observationally consistent.

The developer can use this observation to optimize checking by writing a specification that only specifies the properties that the method relies on. While this approach ensures that the data structure is consistent from the view of a single method, parts of the computation which rely on the actions of multiple methods may be able to observe inconsistencies between the actions of these methods. The developer has two options of how to handle violations once they are detected: the repair algorithm can repair only the portion of the complete specification that the method depends on or the repair algorithm can repair the complete specification.

7.3.3 Incremental Checking

For data structure inconsistencies arising from software faults, the developer can use an incremental approach to checking the consistency constraints. The repair algorithm can use the hardware page protection mechanism to track changes to data structures. We can use this mechanism to incrementally update the affected portions of the model. The consistency constraints that depend on the changed portion of the model are then re-evaluated. This incremental technique enables the repair algorithm to check only the constraints that can be invalidated by the dynamic actions of the application. The downside of using this technique is that the repair algorithm may not notice inconsistencies caused by hardware failures.

A similar incremental approach can be taken with static analysis. A static analysis would analyze the software system to determine which constraints may have been violated since the last consistency check. The specification compiler would use the results of the static analysis to generate repair algorithms that only check the constraints that may have been violated since the last consistency check.

Chapter 8

Developer Control of Repairs

The repair tool contains features designed to provide the developer insight into and control over the repair process. We have included features that provide the developer with the ability to specify preferences for repair actions and even to eliminate undesirable repair actions. The developer can also use the repair dependence graph to gain insight into the potential behaviors of the generated repair algorithms.

8.1 Developer Control

The repair algorithm often has multiple repair options that can be used to satisfy a given constraint; these options may translate into different repaired data structures. Some repair actions may produce more desirable data structures than other repair actions. In particular, some repair actions may produce data structures that lead to acceptable executions while other repair actions may produce data structures that cause the application to perform unacceptably. For example, consider a data structure that consists partially of memory mapped sensor data. Repairs that modify the sensor data would likely be unacceptable, while repairs that adjust the other state to be consistent with the sensor data would likely be acceptable. The developer may be able to provide guidance as to which repair actions are acceptable. 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.

The repair algorithm uses a cost function to decide at runtime how to repair a given constraint. The cost function assigns a cost to satisfying each basic proposition. The repair algorithm uses this cost function to assign costs to each repair option, and then chooses the repair option with the lowest cost. The cost function is implemented as a method that takes a basic proposition or negated basic proposition and returns an integer cost for making that proposition true. The developer may provide an alternate method for computing costs. Suppose that the developer wished to bias the repair algorithm against repairs that falsify inclusion constraints. He or she might write a cost function that assigns a cost of 10 to negated inclusion constraints and a cost of 1 to all other basic propositions. In general, the cost function can be used to bias the repair algorithm towards (or away from) different types of repairs.

In some cases, a given repair action or even set of repair actions should never be performed. For example, suppose the specification compiler generated a repair action that repairs a linked list by deleting it. Our system contains a command line option to exclude satisfying a specific conjunction, and therefore performing any of its model repairs. The mechanism functions by removing these conjunctions from the repair dependence graph. One might also want to exclude data structure updates or compensation updates that are less desirable than other updates. It is straightforward to extend the same mechanism to allow the developer to specify data structure or compensation updates that should not be performed. The specification compiler would then attempt to generate repair algorithms that function without performing these updates or model repairs.

The developer may also wish to eliminate data structure updates, compensation updates, or model repairs that modify certain fields, array elements, sets, or relations. For example, consider an application that reads from a memory-mapped sensor. The repair algorithm should function to make the application's state consistent with the sensor readings and not to make the sensor readings consistent with the application's state. We could easily extend the specification compiler to allow the developer to specify a set of fields, sets, and relations that should not be modified. The specification compiler would eliminate all data structure updates, compensation updates, or model

repairs that may modify these elements.

In some cases, it may be desirable to have complete control over specific repairs. For example, the developer may wish to use domain-specific knowledge to improve the performance of a given class of repairs. While our system does not currently support this, it can easily be extended to support developer provided repair routines. Upon discovering an inconsistency, the repair algorithm would initially call the developer provided repair routine. If the developer provided routine failed, the system could fall back on the automatically-generated code.

In some cases, the repair of certain constraints may not be desirable. Our system can simply detect violations of a given constraint, report the violation, and then stop the execution of the application. The choice of whether to repair or stop when a constraint is violated can be made on a constraint by constraint basis.

Finally, when a model repair requires a new object there are often multiple potential sources for such an object. The repair algorithm can acquire the new object either from a different set or by calling an allocator, and the developer can specify the set or the allocator method. We similarly allow the developer to control the source of objects used to compose tuples in relations.

8.2 Communication with the Developer

In many cases, the developer will treat our specification compiler as a black box that takes in a consistency specification and generates a repair algorithm. However, in some cases the developer may wish to know more about the repair algorithm compilation process. For example, in some cases our specification compiler can fail to generate a repair algorithm for a specification. This can happen for three different reasons:

- In some cases, the specification compiler cannot verify that a repair algorithm terminates. This can occur because either the specification is not satisfiable, the specification compiler is not able to generate a terminating repair algorithm, or in the case of repair algorithms that actually terminate, imprecision in the

termination analysis may prevent the specification compiler from verify that the algorithm terminates.

- The specification compiler can fail to generate a repair algorithm because the specification compiler is unable to generate model repairs for a given basic proposition. For example, the specification compiler does not generate model repairs for the consistency constraint `for s in Set, s.R=s.R*s.R-2` because repairs to the relation `R` change the value of the right hand side of the equality.

- Finally, the specification compiler may be unable to generate data structure updates that implement the model repair.

The developer would like to understand why the specification compiler failed to generate a repair algorithm so that he or she can modify the specification in order to be able to successfully generate a repair algorithm. Even when a repair algorithm is successfully generated, some developers may wish to understand the possible behaviors of the generated repair algorithm.

Our specification compiler generates repair dependence graphs during the compilation process. The developer can examine these graphs to determine what model repairs and updates may be performed by the repair algorithm; which conjunctions will be used to satisfy a constraint; and the dependences between repair actions, the abstract model, and consistency constraints.

The developer can use the repair dependence graph to understand why the specification compiler failed to generate a repair algorithm. The developer can examine the repair dependence graph to determine whether the necessary repair actions were generated for each constraint and examine any cycles in the repair dependence graph to determine why the compiler is unable to generate a terminating repair algorithm from the specification.

Furthermore, since the repair dependence graph enumerates all possible repair actions, the developer can use the repair dependence graph to determine whether the possible behaviors of a generated repair algorithm are acceptable. The developer can use the repair dependence graph to reason about the possible effects of repairs, what

repair actions may be performed as a result of these effects, and how this may affect the data structures. For example, the developer can use the repair dependence graph to determine whether objects may be deleted and which fields may be modified in response to a data structure inconsistency.

Chapter 9

Experience

We next discuss our experience using our repair tool to detect and repair inconsistencies in data structures from five different applications: a word processor, a parallel x86 emulator, an air-traffic control system, a Linux file system, and an interactive game.

9.1 Methodology

We implemented our specification compiler to evaluate our repair algorithm. This implementation consists of approximately 20,800 lines of Java and C code. The implementation compiles specifications into C code that performs the consistency checks and (if necessary) repairs the data structures. Our current implementation is limited in that it does not implement the operations on recursive data structures described in Section 5.5.4 nor does it automatically calculate complexity bounds as described in Chapter 7. The source code for the tool and sample specifications are available at http://www.cag.lcs.mit.edu/∼bdemsky/repair. We ran the applications (with the exception of the parallel x86 emulator) on an IBM ThinkPad X23 with an 866 Mhz Pentium III processor, 384 MB of RAM, and RedHat Linux 8.0.

For each application, we identified important consistency constraints and developed a specification that captured these constraints. To reduce specification overhead, we developed a structure definition extraction tool that uses debugging information in

the executable to automatically generate the structure definitions. This tool works for any application that can be compiled with Dwarf-2 debugging information. We used this tool to automatically generate structure definitions for AbiWord, the parallel x86 emulator, CTAS, and Freeciv. We also obtained a workload that caused the application to generate corrupt data structures. When possible, the workload triggered a known programming error. In other cases, we used fault insertion to mimic either the effect of a previously corrected programming error or a common data structure inconsistency source. We then compared the results of running a chosen workload with and without inconsistency detection and repair.

Limitations of the Study We designed this study to test if data structure repair, when given a specification that detects an inconsistency, effectively enables applications to recover from that inconsistency. We verified that a selected software error (or fault injection) caused the application to fail, and then wrote the relevant consistency specifications for the effected data structure. The study was not designed to evaluate how difficult it is to write complete data structure consistency specifications for an application, or to evaluate how likely specifications written by developers are to discover inconsistencies caused by real world errors.

9.2 AbiWord

AbiWord is a full-featured, open source word processing application available at www.abisource.com. It consists of over 360,000 lines of C++ code. It can import and export many file formats including Microsoft Word documents.

AbiWord uses a piece table data structure to internally represent documents. Figure 9-1 shows a piece table data structure. The piece table contains a doubly-linked list of document fragments. A consistent piece table contains a reference to both the head and the tail of the doubly linked list of document fragments. A consistent fragment contains a reference to the next fragment in the list and a reference to the previous fragment in the list. Furthermore, a consistent list of fragments contains

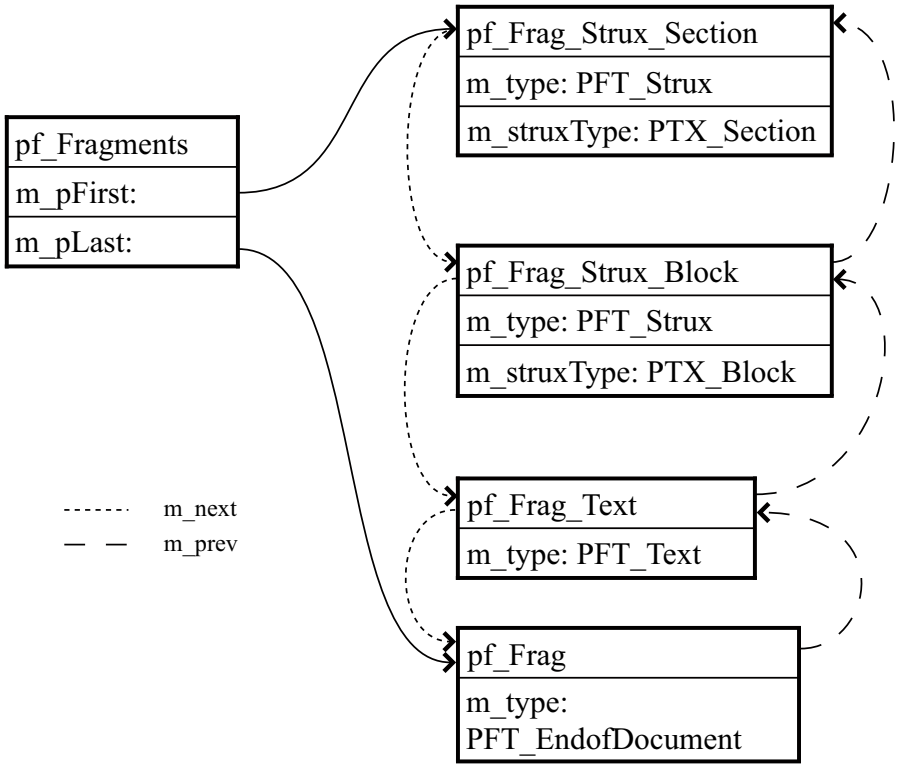


Figure 9-1: Piece Table Data Structure

both a section fragment and a paragraph fragment.

9.2.1 Specification

We developed a specification for the piece table data structure. Our specification consists of 94 lines, of which 70 contain automatically-generated structure definitions. Figure 9-2 gives the structure definitions for the data structures that make up the piece table. The `pf_Fragments` data structure maintains pointers to the beginning and end of the linked list of document fragments. The `pf_Frag` structure definition gives the generic layout of document fragments, the `pf_Frag_Strux` structure definition gives the layout of structural document fragments, the `pf_Frag_Strux_Block` structure definition gives the layout of the block fragment, and the `pf_Frag_Strux_Section` structure definition gives the layout of the section fragment. Figure 9-3 gives the set and relation declarations for the abstract model of the piece table data structure.

Figure 9-4 gives the model definition rules that define this abstract model in terms of the concrete data structures. The first rule adds the `pf_Fragments` object to the `Fragments` set. The next two rules add the section and block fragments to the `firstblock` and `secondblock` sets, respectively. The fourth, fifth, and sixth rules create the set of fragments and the `Next` and `Prev` relations that link these fragments. The remaining rules abstract the length field and the last fragment in the list. Figure 9-5 gives the model constraints for the piece table. [1] These constraints ensure that the piece table exists, that it contains a section fragment, that it contains a paragraph fragment, that the `m_prev` pointers are consistent with the `m_next` pointers, that the `m_pLast` pointer points to the end of the fragment list, and that each fragment has a valid `m_length` field.

9.2.2 Error

An error in version 0.9.5 (and previous versions) of AbiWord causes AbiWord to attempt to append text to a piece table which lacks a section fragment or a paragraph fragment. Importing certain valid Microsoft Word documents triggers this error, causing AbiWord to fail with a segmentation violation when the user attempts to load the document. This error was fixed in subsequent versions of AbiWord by placing explicit checks for the section and paragraph fragments in the code that appends text to the piece table. If these checks indicate that a fragment is missing, the patch creates the appropriate section and adds it to the piece table.

We obtained a Microsoft Word document that triggered this error and used our system to enhance AbiWord with data structure repair as described in this dissertation. It is important to note that this document is a valid Microsoft Word document, and that Microsoft Word has no problem loading the document.

[1] Our implementation supports using $\sim$`R` to indicate the inverse of the relation `R`. Therefore, the expression `f.`$\sim$`Next` denotes the image of `f` under the inverse of the `Next` relation.

```
structure UT_Vector {
   void * m_pEntries;
   int m_iCount;
   int m_iSpace;
   int m_iCutoffDouble;
   int m_iPostCutoffIncrement;
}

structure pf_Fragments {
   pf_Frag * m_pFirst;
   pf_Frag * m_pLast;
   UT_Vector m_vecFrags;
   byte m_bAreFragsClean;
   reserved byte[3];
   pf_Frag * m_pCache;
}

structure pf_Frag {
   void * _vptr_pf_Frag;
   int m_type;
   int m_length;
   pf_Frag * m_next;
   pf_Frag * m_prev;
   fd_Field * m_pField;
   pt_PieceTable * m_pPieceTable;
   int m_docPos;
}

structure pf_Frag_Strux
  subclass of pf_Frag {
   void * _vptr_pf_Frag;
   int m_type;
   int m_length;
   pf_Frag * m_next;
   pf_Frag * m_prev;
   fd_Field * m_pField;
   pt_PieceTable * m_pPieceTable;
   int m_docPos;
   int m_struxType;
   int m_indexAP;
   UT_Vector m_vecFmtHandle;
}

structure pf_Frag_Strux_Block
  subclass of pf_Frag_Strux {
   void * _vptr_pf_Frag;
   int m_type;
   int m_length;
   pf_Frag * m_next;
   pf_Frag * m_prev;
   fd_Field * m_pField;
   pt_PieceTable * m_pPieceTable;
   int m_docPos;
   int m_struxType;
   int m_indexAP;
   UT_Vector m_vecFmtHandle;
}

structure pf_Frag_Strux_Section
  subclass of pf_Frag_Strux {
   void * _vptr_pf_Frag;
   int m_type;
   int m_length;
   pf_Frag * m_next;
   pf_Frag * m_prev;
   fd_Field * m_pField;
   pt_PieceTable * m_pPieceTable;
   int m_docPos;
   int m_struxType;
   int m_indexAP;
   UT_Vector m_vecFmtHandle;
}

pf_Fragments * fragments;
```

Figure 9-2: Structure Definitions for AbiWord (extracted)

```
set Fragments(pf_Fragments);
set Frags(pf_Frag): firstblock | secondblock | Last;
set Last(pf_Frag);
set firstblock(pf_Frag_Strux_Section);
set secondblock(pf_Frag_Strux_Block);
Next: Frags -> Frags;
Prev: Frags -> Frags;
set LFrag(pf_Frag);
PLast: Fragments -> LFrag;
Length: Frags -> int;
```

Figure 9-3: Model Declarations for AbiWord

```
!(fragments=null) => fragments in Fragments;
for f in Fragments, ((!(f.m_pFirst=null)) and (f.m_pFirst.m_type=2))
  and cast(pf_Frag_Strux,f.m_pFirst).m_struxType=0 =>
  cast(pf_Frag_Strux_Section,f.m_pFirst) in firstblock;
for f in firstblock, ((!(f.m_next=null)) and (f.m_next.m_type=2))
  and cast(pf_Frag_Strux,f.m_next).m_struxType=1 =>
  cast(pf_Frag_Strux_Block,f.m_next) in secondblock;
for f in Frags, !(f.m_next=null) => f.m_next in Frags;
for f in Frags, !(f.m_next=null) => <f,f.m_next> in Next;
for f in Frags, !(f.m_prev=null) => <f,f.m_prev> in Prev;
for f in Frags, (f.m_next=null) => f in Last;
for f in Frags, true => <f,f.m_length> in Length;
for f in Fragments, !(f.m_pLast=null) => <f,f.m_pLast> in PLast;
for f in Fragments, !(f.m_pLast=null) => f.m_pLast in LFrag;
```

Figure 9-4: Model Definition Rules for AbiWord

```
size(firstblock)=1;
size(Fragments)=1;
size(secondblock)=1;
for f in Frags, (!(size(f.Next)=1)) or ((f.Next.Prev=f) and
  size(f.Next.Prev)=1);
for f in Frags, (size(f.~Next)<=1);
for f in firstblock, size(f.~Next)=0;
for l in Last, for f in Fragments], f.PLast=l and size(f.PLast)=1;
for f in Frags, f.Length>=1;
```

Figure 9-5: Consistency Constraints for AbiWord

9.2.3 Results

We began by running a version of AbiWord without repair. We imported a Microsoft Word document that was known to crash AbiWord. This document caused AbiWord to append text to an (inconsistent) empty fragment list. Later when Abiword attempts to process this internal representation of the document, it crashes due to this inconsistency. We repeated the experiment with a version of AbiWord that incorporates a repair algorithm generated with our specification compiler. We imported a Microsoft Word document that was known to crash AbiWord. When AbiWord executes the text flush method to append a text fragment, the application invokes the generated repair algorithm, then the generated repair algorithm detects that the piece table is missing a section and paragraph fragment and repairs the inconsistency by adding a section fragment and a paragraph fragment. To perform this repair, the generated repair algorithm first allocates the section fragment. It then redirects the `m_pFirst` pointer to the newly allocated section fragment. It then allocates a paragraph fragment, and redirects the `m_next` pointer of the section fragment to the newly allocated paragraph fragment. Finally, it points the `m_prev` field of each fragment to the previous fragment, and redirects the `m_pLast` pointer to the end of the fragment list. The result of this repair is that AbiWord is able to successfully append the text to the list. The effect of this repair on the end user is that he or she is able to successfully load and edit Word documents. This particular combination of error and repair strategy does not result in the loss of any information in the document. However, it is possible that other errors could result in repairs that lose information.

For our benchmark file, a consistency check for AbiWord took 0.06 milliseconds; performing a check and repair took 0.55 milliseconds. As the check is only performed when text is flushed to the piece table (a handful of times), the checking overhead is not significant. For our test document, the repair algorithm performed 7 data structure updates and rebuilt the abstract model 10 times.

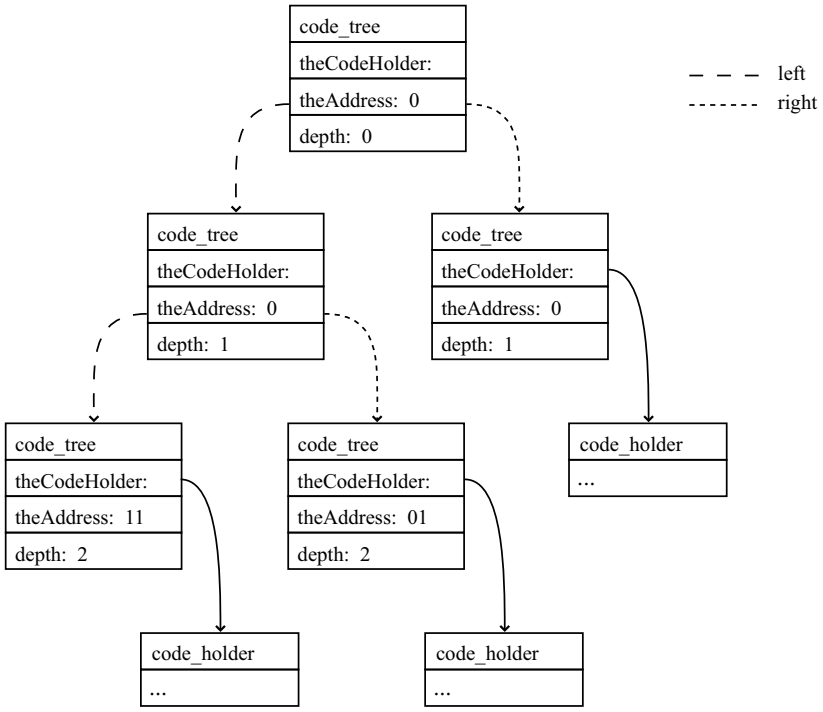


Figure 9-6: Cache Data Structure

9.3 Parallel x86 emulator

The MIT RAW [35] architecture consists of a grid of interconnected computational tiles. Each of these tiles is a MIPS-style processor. The MIT RAW group developed a software-based parallel x86 emulator to explore running x86 binaries on this architecture. The x86 emulator uses a tree data structure to cache translations of the x86 code. Figure 9-6 shows an example of the tree data structure used for caching translations. To efficiently manage the size of the cache, the emulator maintains a variable that stores the current size of the cache.

9.3.1 Specification

We developed a specification that ensures that the computed size of the cache is correct. Our specification consists of 110 lines, of which 90 contain structure definitions that were automatically extracted. Figures 9-7, 9-8, 9-9, and 9-10 give the structure definitions, model declarations, model definition rules, and model constraints, respectively. The specification ensures that the computed size of the cache is consistent with the actual size of the cache.

9.3.2 Error

The x86 emulator uses a tree to cache translated x86 instructions. A pre-existing error in the tree insertion method causes (under some conditions) the cache management code to add the size of the inserted cache item to this variable twice. When this item is removed, the code subtracts the size of the item only once. The net result of inserting and removing such an item is that the computed size of the cache becomes increasingly larger than the actual size of the cache. The end result is that the emulator eventually crashes when it attempts to remove items from an empty cache.

9.3.3 Results

Our test workload ran gzip on the x86 emulator. We instrumented the x86 emulator to call the generated repair code prior to any cache eviction. The generated repair algorithm computes the size of cache data structure and detects an inconsistency between the size it computes and the stored value. To repair this inconsistency, the repair algorithm updates the stored size of the cache to be equal to the actual size of the cache. The result of this repair is that the x86 emulator no longer attempts to evict cache items from an empty cache. The effect of this repair on the end user is that the x86 emulator is able to successfully execute gzip. Without repair, the emulator stops with a failed assertion. With repair, the emulator successfully executes gzip.

In general, the repair algorithm repairs data structures by assigning fields to new values that make the data structure consistent, but it may not generate the same

```
structure TYPE2 {
   byte opcode;
   byte special;
   byte common;
   byte regImm;
   byte resultRegister;
   byte inputARegister;
   byte inputBRegister;
   byte pad0;
   int immediate;
   int shiftAmount;
   int binaryRepresentation;
   byte hasSpecial;
   byte hasCommon;
   byte hasRegImm;
   byte writesResultRegister;
   byte readsResultRegister;
   byte readsInputARegister;
   byte readsInputBRegister;
   byte readsSPRs;
   byte writesSPRs;
   byte readsMemory;
   byte writesMemory;
   byte hasImmediate;
   byte hasShiftAmount;
   byte changesControlFlow;
   byte writesHILO;
   byte readsHILO;
}

structure code_holder {
   int addressOfCode;
   _UCodeBlock * theUCodeBlock;
   int ucodeLevel;
   TYPE0 * theRawInstructionBlockDense;
   int rawCodeLevel;
   int lockMutex;
}

structure code_tree {
   code_tree * left;
   code_tree * right;
   code_holder * theCodeHolder;
   int theAddress;
   int depth;
}
```

```
structure TYPE0 {
   int * translatedCode;
   TYPE2 * instructions;
   int numberOfRawInstructions;
}

structure TYPE1 {
   int lit32;
   short val1;
   short val2;
   short val3;
   byte opcode;
   byte size;
   byte flags_r;
   byte flags_w;
   byte tag1;
   byte tag2;
   byte tag3;
   byte extra4b;
   byte cond;
   byte signed_widen;
   int jmpkind;
   byte argc;
   byte regparms_n;
   byte has_ret_val;
   int regs_live_after;
}

structure _UCodeBlock {
   int orig_eip;
   int used;
   int size;
   int numberOfNextIPs;
   int nextIP[3];
   int nextIPType[3];
   byte deadFlagsOut;
   byte deadFlagsIn;
   reserved byte[2];
   TYPE1 * instrs;
   int nextTemp;
}

code_tree * root;
int cacheSize;
```

Figure 9-7: Structure Definitions for x86 Emulator (extracted)

```
set TreeNodes(code_tree);
set CodeHolders(code_holder);
set UCodeBlocks(_UCodeBlock);
set InstructionBlocks(TYPE0);
NumRawInstructions:InstructionBlocks->int;
NumTranslatedInstructions:InstructionBlocks->int;
Size:UCodeBlocks->int;
set Global(int);
CacheSize:Global->int;
```

Figure 9-8: Model Declarations for x86 Emulator

```
root!=null => root in TreeNodes;
for n in TreeNodes, n.left!=null => n.left in TreeNodes;
for n in TreeNodes, n.right!=null => n.right in TreeNodes;
for n in TreeNodes, n.theCodeHolder!=null => n.theCodeHolder in CodeHolders;
for n in CodeHolders, n.theUCodeBlock!=null => n.theUCodeBlock in UCodeBlocks;
for n in UCodeBlocks, true => <n,n.size> in Size;
for n in CodeHolders, n.theRawInstructionBlockDense!=null =>
  n.theRawInstructionBlockDense in InstructionBlocks;
for n in InstructionBlocks, true =>
  <n,n.numberOfRawInstructions> in NumRawInstructions;
for n in InstructionBlocks, n.translatedCode!=null =>
  <n,n.numberOfRawInstructions> in NumTranslatedInstructions;
true => 0 in Global;
for n in Global, true => <n, cacheSize> in CacheSize;
```

Figure 9-9: Model Definition Rules for x86 Emulator

```
for n in Global, n.CacheSize=size(TreeNodes)*20+size(CodeHolders)*24+
  size(UCodeBlocks)*52+sum(UCodeBlocks.Size)*20+size(InstructionBlocks)*12+
  sum(InstructionBlocks.NumRawInstructions)*24+
  sum(InstructionBlocks.NumTranslatedInstructions)*4;
```

Figure 9-10: Consistency Constraints for x86 Emulator

data structure that a correct execution would have. However, in some cases applications maintain redundant information in data structures. Developers typically use redundant information for efficiency reasons. This enables the application to avoid recomputing some expression. In this case of the x86 emulator, the repair algorithm is able to use redundant information to recompute the size of the cache, and then to replace the incorrect value with the correct value.

9.4 CTAS

The Center-TRACON Automation System (CTAS) is a set of air-traffic control tools developed at the NASA Ames research center [1, 34]. The system is designed to help air traffic controllers visualize and manage the complex air traffic flows at centers surrounding large metropolitan airports. [2] In addition to graphically displaying the location of the aircraft within the center, CTAS also uses sophisticated algorithms to predict aircraft trajectories and schedule aircraft landings. The goal is to automate much of the aircraft traffic management, reducing traffic delays and increasing safety. The current source code consists of over 1 million lines of C and C++ code. Versions of this source code are deployed at seven of the 21 centers in the continental United States (Dallas/Ft. Worth, Los Angeles, Denver, Miami, Minneapolis/St. Paul, Atlanta, and Oakland) and are in daily use at these centers.

Strictly speaking, CTAS is an advisory system in that the air-traffic controllers are expected to be able to bring aircraft down safely even if the system fails. Nevertheless, CTAS has several properties that are characteristic of our set of target applications. Specifically, it is a central part of a broader system that manages and controls safety-critical real-world phenomena and, as is typical of these kinds of systems, it deals with a bounded window of time surrounding the current time.

The CTAS software maintains data structures that store aircraft data. Our ex-

[2] A center is a geographical region surrounding the major airport. In addition to the major airport, each center typically contains several smaller regional airports. Because these airports share overlapping airspaces, the air traffic flows must be coordinated for all of the aircraft within the center, regardless of their origin or destination.

Flight_plan_st:
category: 2 (ARRIVAL)
origin: -999999 (NOT SET)
destination: 0 (DFW airport)
...

Figure 9-11: CTAS Data Structure

periments focus on the flight plan objects, which store the flight plans for the aircraft currently within the center. Figure 9-11 shows a sample CTAS flight plan data structure. These flight plan data structures contain both an origin and destination airport identifier. The software uses these identifiers as indices into an array of airport data structures. Flight plans are transmitted to CTAS as a long character string. The structure of this string is somewhat complicated, and parsing the flight plan string is a challenging activity.

9.4.1 Specification

Our specification captures the constraint that the flight plan indices must be within the bounds of the airport data array. The specification itself consists of 101 lines, of which 84 lines contain structure definitions. The primary challenge in developing this specification was reverse engineering the source to develop an understanding of the data structures. Once we understood the data structures, developing the specification was straightforward.

The specification ensures that aircraft are assigned to a valid category, that departing aircraft are assigned to a valid origin airport, that arriving aircraft are assigned to

a valid destination airport, and that if an origin or a destination airport is unassigned that the corresponding field is set to the special "not set" value.

9.4.2 Fault Injection

Flight plan parsing is surprisingly difficult. Part of the reason for this is that the format of the flight plan strings was originally developed for human flight controllers to use. Earlier versions of CTAS contained a buffer overrun error in the flight plan parsing code that could result in corrupted flight plans. The source of this error is that flight plan strings could be longer than what the CTAS designers had thought, and as a result CTAS could overwrite other fields in the flight plan data structure. Our fault insertion methodology attempts to mimic errors in the flight plan processing that produce illegal values in the flight plan data structures. Our fault injector corrupts the destination field of the flight plan. When the application uses these corrupted values to access the array of airport data, the array access is out of bounds, which typically leads to the application failing because of an addressing error.

9.4.3 Results

We used a recorded midday radar feed from the Dallas-Ft. Worth center as a workload. The generated repair algorithm is invoked whenever a new flight plan enters the system or an old flight is amended. Without repair, CTAS fails because of an addressing exception. With repair, it continues to execute in a largely acceptable state. Specifically, the effect of the repair is to potentially change the origin or destination airport of the aircraft with the faulty flight plan. Even with this change, continued operation is clearly a better alternative than failing. First, one of the primary purposes of the system, visualizing aircraft flow, is unaffected by the repair. Second, only the origin or destination airport of the plane whose flight plan triggered the error is affected. All other aircraft are processed with no errors at all.

Rebooting CTAS after a crash is an inadequate solution. After a reboot, CTAS takes several minutes to reacquire flight plans and radar data. Furthermore, as long

as the problematic flight plan remains in the system, rebooting the system will be ineffective: the system will reacquire the same problematic flight plan, reprocess it, and fail again for the same reason.

On our benchmark system, an average consistency check for CTAS took 0.07 milliseconds and performing a check and repair took 0.15 milliseconds. As the check is only performed when a flight plan is changed, the checking overhead is not significant. For our test run, the repair algorithm performed 1 data structure update and rebuilt the abstract model 3 times.

9.5 Freeciv

Freeciv is an interactive, multi-player game available at www.freeciv.org. The Freeciv server maintains a map of the game world. Figure 9-12 show a sample map data structure. This map references an array of tiles. Each tile in this array has a terrain value chosen from a set of legal terrain values. Additionally, cities may be placed on the tiles (indicated by a reference from the tile to a city object).

9.5.1 Specification

The specification consists of 191 lines, of which 173 lines contain structure definitions that were automatically extracted. The principal challenge in developing this specification was reverse engineering the Freeciv source (which consists of 73,000 lines of C code) to develop an understanding of the data structures. Once we understood the data structures, developing the specification was straightforward.

Figures 9-13, 9-14, 9-15, 9-16, and 9-17 give part one of the structure definitions, part two of the structure definitions, the model declarations, the model definition rules, and the consistency constraints, respectively. The specification ensures that the game map exists, that the grid of tiles exists, that each tile has a legal terrain value, and that cities are not placed in the ocean.

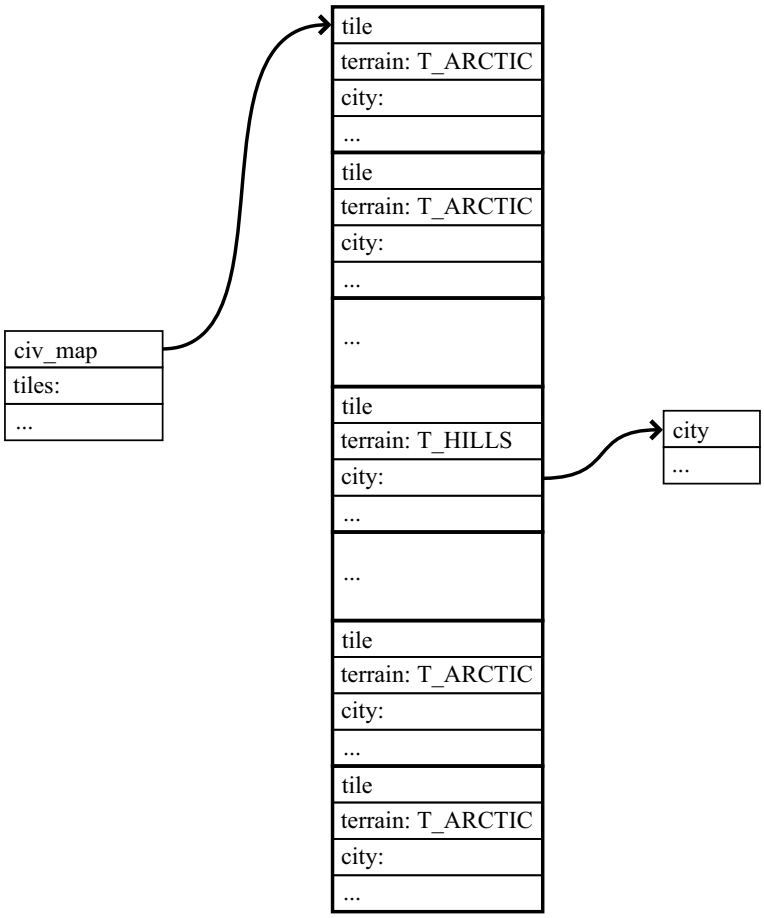


Figure 9-12: Freeciv Data Structure

9.5.2 Fault Injection

Our fault injection strategy changes the terrain values in pseudo-randomly selected tiles 35 times during the execution of the application. There are two possible errors: illegal terrain values or cities located on an ocean tile instead of a land tile. Our repair algorithm repairs these kinds of errors by assigning a legal terrain value to any tile with an illegal value and by assigning a land terrain value to any ocean tiles that contain a city.

```
structure civ_map {
   int xsize;
   int ysize;
   int seed;
   int riches;
   int is_earth;
   int huts;
   int landpercent;
   int grasssize;
   int swampsize;
   int deserts;
   int mountains;
   int riverlength;
   int forestsize;
   int generator;
   int tinyisles;
   int separatepoles;
   int num_start_positions;
   int fixed_start_positions;
   int have_specials;
   int have_huts;
   int have_rivers_overlay;
   int num_continents;
   tile_array * tiles;
   map_position start_positions[63];
}

structure tile_array {
  tile elem[m.xsize*m.ysize];
}

structure tile {
   int terrain;
   int special;
   city * city;
   unit_list units;
   int known;
   int sent;
   int assigned;
   city * worked;
   short continent;
   byte move_cost[8];
   reserved byte[2];
}

structure unit_list {
   genlist list;
}
```

```
structure genlist {
   int nelements;
   genlist_link null_link;
   genlist_link * head_link;
   genlist_link * tail_link;
}

structure genlist_link {
   genlist_link * next;
   genlist_link * prev;
   void * dataptr;
}

structure map_position {
   int x;
   int y;
}

structure athing {
   void * ptr;
   int size;
   int n;
   int n_alloc;
}

structure ceff_vector {
   athing vector;
}

structure worklist {
   int is_valid;
   byte name[32];
   int wlefs[16];
   int wlids[16];
}

structure ai_choice {
   int choice;
   int want;
   int type;
}
```

Figure 9-13: Part 1 of the Structure Definitions for Freeciv (extracted)

```
structure city {
   int id;
   int owner;
   int x;
   int y;
   byte name[32];
   int size;
   int ppl_happy[5];
   int ppl_content[5];
   int ppl_unhappy[5];
   int ppl_angry[5];
   int ppl_elvis;
   int ppl_scientist;
   int ppl_taxman;
   int trade[4];
   int trade_value[4];
   int food_prod;
   int food_surplus;
   int shield_prod;
   int shield_surplus;
   int trade_prod;
   int corruption;
   int tile_trade;
   int shield_bonus;
   int tax_bonus;
   int science_bonus;
   int luxury_total;
   int tax_total;
   int science_total;
   int food_stock;
   int shield_stock;
   int pollution;
   int incite_revolt_cost;
   int is_building_unit;
   int currently_building;
   byte improvements[200];
   ceff_vector effects;
   worklist worklist;
   int city_map[25];
   unit_list units_supported;
   int steal;
   int did_buy;
   int did_sell;
   int is_updated;
   int turn_last_built;
   int turn_changed_target;
   int changed_from_id;
   int changed_from_is_unit;
   int disbanded_shields;
   int caravan_shields;
   int before_change_shields;
   int anarchy;
   int rapture;
   int was_happy;
   int airlift;
   int original;
   int city_options;
   int synced;
   unit_list info_units_supported;
   unit_list info_units_present;
   ai_city ai;
}

structure ai_city {
   int workremain;
   int ai_role;
   int building_want[200];
   int danger;
   int diplomat_threat;
   int urgency;
   int grave_danger;
   int wallvalue;
   int trade_want;
   ai_choice choice;
   int downtown;
   int distance_to_wonder_city;
   short detox[25];
   short derad[25];
   short mine[25];
   short irrigate[25];
   short road[25];
   short railroad[25];
   short transform[25];
   short tile_value[25];
   int settler_want;
   int founder_want;
   int a;
   int f;
   int invasion;
}

civ_map *map;
```

Figure 9-14: Part 2 of the Structure Definitions for Freeciv (extracted)

```
set MAP(civ_map);
set GRID(tilegrid);
set TILE(tile);
set TERRAINTYPES(int);
set CITY(city);
CITYMAP: TILE -> CITY;
TERRAIN: TILE -> int;
```

Figure 9-15: Model Declarations for Freeciv

```
true => map in MAP;
for m in MAP, true => m.tiles in GRID;
for t in GRID, for x=0 to map.xsize-1, for y=0 to map.ysize-1], true =>
  t.elem[x+(y*map.xsize)] in TILE;
for t in TILE, true => <t,t.terrain> in TERRAIN;
for t in TILE, !(t.city = NULL) => <t,t.city> in CITYMAP;
for t in TILE, !(t.city = NULL) => t.city in CITY;
```

Figure 9-16: Model Definition Rules for Freeciv

```
size(MAP)=1;
size(GRID)=1;
for t in TILE, t.TERRAIN>=0 and t.TERRAIN<=11;
for c in CITY, size(c.~CITYMAP)=1;
for c in CITY, !c.~CITYMAP.TERRAIN=7;
```

Figure 9-17: Consistency Constraints for Freeciv

9.5.3 Results

Freeciv comes with a built-in test mode in which several automated players play against each other. Our workload simply runs the application in this built-in test mode. We configured the map to contain 4,000 tiles. With repair, the game was able to execute without failing on our test run (although the game played out differently because of changed terrain values). Freeciv is played in rounds. The generated repair algorithm was invoked at the beginning of each round of the game. The generated repair algorithm detected inconsistent terrain values and cities placed on ocean tiles, then reset the inconsistent terrain fields to legal terrain values. Without repair, the game crashed with a segmentation fault caused by indexing an array with an illegal terrain value.

On our benchmark system, an average consistency check for Freeciv took 3.62 milliseconds and performing a check and repair took 15.66 milliseconds. As the check is only performed once per a game round, the checking overhead is not significant. For our test run, repair algorithm performed 1.8 data structure updates and rebuilt the abstract model 4.6 times on average.

9.6 A Linux File System

Our Linux file system application implements a simplified version of the Linux ext2 file system. The file system, like other Unix file systems, contains bitmaps that identify free and used disk blocks. The file system uses these disk blocks to support fast disk block and inode allocation operations. For our experiments, we used a file system with 1024 disk blocks.

9.6.1 Specifications

Our consistency specification contains 108 lines, of which 55 lines contain structure definitions. Because the structure of such file systems is widely documented in the literature, it was relatively easy for us to develop the specification.

```
structure Block {
     reserved byte[d.s.blocksize];
}

structure Disk {
     Block  b[d.s.NumberofBlocks];
     label b[0]: Superblock s;
     label b[1]: Groupblock g;
}

structure Superblock subtype of Block {
     int FreeBlockCount;
     int FreeInodeCount;
     int NumberofBlocks;
     int NumberofInodes;
     int RootDirectoryInode;
     int blocksize;
}

structure Groupblock subtype of Block {
     int BlockBitmapBlock;
     int InodeBitmapBlock;
     int InodeTableBlock;
     int GroupFreeBlockCount;
     int GroupFreeInodeCount;
}
```

```
structure InodeTable subtype of Block {
     Inode itable[d.s.NumberofInodes];
}

structure InodeBitmap subtype of Block {
     bit inodebitmap[d.s.NumberofInodes];
}

structure BlockBitmap subtype of Block {
     bit blockbitmap[d.s.NumberofBlocks];
}

structure Inode {
     int filesize;
     int Blockptr[12];
     int referencecount;
}

structure DirectoryBlock subtype of Block {
     DirectoryEntry de[d.s.blocksize/128];
}

structure DirectoryEntry {
     byte name[124];
     int inodenumber;
}

Disk *d;
```

Figure 9-18: Structure Definitions for File System

Figures 9-18, 9-19, 9-20, and 9-21 give the structure definitions, the model declarations, the model definition rules, and the consistency constraints, respectively. The specification ensures that the inode table, inode bitmap, block bitmap, and root directory inode all exist; that the inode bitmap correctly indicates whether an inode is used; that the block bitmap correctly indicates whether a block is used; that the inode reference count is correct; that a file's size is consistent with the number of blocks it has; and that a block is only in at most one file or directory.

9.6.2 Fault Injection

Our fault insertion mechanism for this application simulates the effect of a system crash or power failure: it shuts down the file system (potentially in the middle of an operation that requires several disk writes), then discards the cached state. Our

```
set Block(Block): UsedBlock | FreeBlock;
set FreeBlock(Block);
set Inode(Inode): UsedInode | FreeInode;
set FreeInode(Inode);
set UsedInode(Inode): FileInode | DirectoryInode;
set FileInode(Inode);
set DirectoryInode(Inode): RootDirectoryInode;
set RootDirectoryInode(Inode);
set UsedBlock(Block): SuperBlock | GroupBlock | FileDirectoryBlock |
  InodeTableBlock | InodeBitmapBlock | BlockBitmapBlock;
set FileDirectoryBlock(Block): DirectoryBlock | FileBlock;
set SuperBlock(Superblock);
set GroupBlock(Groupblock);
set FileBlock(Block);
set DirectoryBlock(Block);
set InodeTableBlock(InodeTable);
set InodeBitmapBlock(InodeBitmap);
set BlockBitmapBlock(BlockBitmap);
set DirectoryEntry(DirectoryEntry);
inodeof: DirectoryEntry -> UsedInode;
contents: UsedInode -> FileDirectoryBlock;
inodestatus: Inode -> int;
blockstatus: Block -> int;
referencecount: Inode -> int;
filesize: Inode -> int;
```

Figure 9-19: Model Declarations for File System

```
true => d.s in SuperBlock;
true => d.g in GroupBlock;
true => cast(InodeTable,d.b[d.g.InodeTableBlock]) in InodeTableBlock;
true => cast(InodeBitmap,d.b[d.g.InodeBitmapBlock]) in InodeBitmapBlock;
true => cast(BlockBitmap,d.b[d.g.BlockBitmapBlock]) in BlockBitmapBlock;
for itb in InodeTableBlock, true =>
  itb.itable[d.s.RootDirectoryInode in RootDirectoryInode;
for itb in InodeTableBlock, for j=0 to d.s.NumberofInodes-1],
  !(itb.itable[j] in? UsedInode) => itb.itable[j] in FreeInode;
for j=0 to d.s.NumberofBlocks-1, !(d.b[j] in? UsedBlock) => d.b[j] in FreeBlock;
for di in DirectoryInode, for j=0 to ((d.s.blocksize/128)-1), for k=0 to 11,
  true => cast(DirectoryBlock,d.b[di.Blockptr[k]]).de[j] in DirectoryEntry;
for i in UsedInode, for j=0 to 11, !(i.Blockptr[j]=0) =>
  <i,d.b[i.Blockptr[j]]> in contents;
for i in UsedInode, for j=0 to 11, !(i.Blockptr[j]=0) =>
  d.b[i.Blockptr[j]] in FileBlock;
for j=0 to d.s.NumberofInodes-1, for itb in InodeTableBlock,
  for ibb in InodeBitmapBlock, true =>
  <itb.itable[j], ibb.inodebitmap[j]> in inodestatus;
for de in DirectoryEntry,for itb in InodeTableBlock, !(de.inodenumber = 0) =>
  itb.itable[de.inodenumber] in FileInode;
for de in DirectoryEntry,for itb in InodeTableBlock, !(de.inodenumber = 0) =>
  <de, itb.itable[de.inodenumber]> in inodeof;
for j in UsedInode, true => <j,j.referencecount> in referencecount;
for j in UsedInode, true => <j,j.filesize> in filesize;
for j=0 to d.s.NumberofBlocks-1, for bbb in BlockBitmapBlock, true =>
  <d.b[j],bbb.blockbitmap[j]> in blockstatus;
```

Figure 9-20: Model Definition Rules for File System

```
size(InodeTableBlock)=1;
size(InodeBitmapBlock)=1;
size(BlockBitmapBlock)=1;
size(RootDirectoryInode)=1;
for u in UsedInode, u.inodestatus=true;
for f in FreeInode, f.inodestatus=false;
for u in UsedBlock, u.blockstatus=true;
for f in FreeBlock, f.blockstatus=false;
for i in UsedInode, i.referencecount=size(i.~inodeof);
for i in UsedInode, i.filesize <= size(i.contents)*8192;
for b in FileDirectoryBlock,size(b.~contents)=1;
```

Figure 9-21: Consistency Constraints for File System

workload opens and writes several files, closes the files, then reopens the files to verify that the data was written correctly. To apply our fault insertion strategy to this workload, we crash the system part of the way through writing the files, then rerun the workload. The second run of the workload overwrites the partially written files and checks that the final versions are correct.

Possible sources of errors include incorrect bitmap blocks (caused by discarding correct cached versions) and incomplete file system operations that leave the disk image in an inconsistent state. Specifically, incomplete removal and hardlink creation operations may leave inodes with incorrect reference counts, and incomplete open operations that create new files may leave directory and inode entries in incorrectly initialized states. The repair algorithm first traverses the blocks and inodes in the file system to construct a model of the file system. It then uses the model to compute correct values for the bitmap blocks and reference counts.

9.6.3 Results

In our tested cases, the algorithm is able to repair the file system and the workload correctly runs to completion. The generated repair algorithm was invoked immediately before the application opens the file system. The generated repair algorithm detects inconsistencies between the block and inode bitmaps and the actual use of the blocks and inodes. It uses the reachability of the blocks and inodes to generate the correct values for the block and inode bitmaps. It then updates these bitmaps with the new correct values.

Without repair, files end up sharing inodes and disk blocks. As a result, the file contents are incorrect. In addition to repairing the errors introduced by our failure insertion strategy, our tool is also able to allocate and rebuild the blocks containing the inode and block allocation bitmaps, allocate a new inode table block, and allocate a new inode for the root directory. The repair algorithm is limited in that if the entries describing aspects of basic file system format (such as the size of the blocks) become corrupted, the tool may fail to correctly repair the file system.

Fsck [5] is a hand-coded file system check and repair utility for the Unix file

system. It performs the same basic task, but since it is hand-coded for this specific task it takes advantage of the developer's domain specific knowledge of the problem. For example, fsck can take advantage of backup copies of the superblock, it stores unlinked blocks and inodes as files in the lost+found directory, and replicates shared blocks. As a result, fsck can recover information that our automatically-generated repair algorithm may lose. However, testing and debugging fsck was likely challenging, and various implementations of fsck have contained errors [4, 2].

9.7 Randomized Fault Injection

In addition to the fault injection experiments previously described, we performed a more extensive randomized fault injections on two of the benchmarks, Freeciv and CTAS. These experiments were designed to understand the effectiveness of our data structure repair technique.

For each experimental run, we injected a single (randomly generated) fault in a data structure and then monitored whether the application crashed after the fault was injected. We then repeated the execution multiple times to help us understand what percentage of the injected faults our repair algorithm enabled the application to tolerate.

9.7.1 CTAS Results

Our fault injector for CTAS selected a flight plan, a field (either the `category`, `destination`, or `origin` field) to corrupt, and a corrupt value (between -5,000 and 5,000) to assign to the field. We then ran 1,000 trials with the randomized fault injector with both a version of CTAS that incorporates a repair algorithm and a version of CTAS without a repair algorithm. We declared a particular run of CTAS a success as soon as the 1500th flight plan entered the system.

Figure 9-22 presents the results for the CTAS benchmark. The crashes we observed without repair were due to segmentation faults. From these results, we can see that the generated repair algorithm enabled CTAS to recover from all of the injected

errors. Notice that many of the injected faults do not cause either version to crash. This can happen because CTAS does not use the value, the value does not cause CTAS to perform an illegal operation, or the faulty value does not violate any of the consistency properties that CTAS expects from the flight plan data structure.

Version	Number of Successful Runs (out of 1,000)
Repair	1,000
No Repair	715

Figure 9-22: Results of Randomized Fault Injection for CTAS

9.7.2 Freeciv Results

Our fault injector for Freeciv selected a game round, a game tile, and a corrupt value (between -5,000 and 5,000) to assign to the field. We ran 10,000 trials with the randomized fault injector on both a version of Freeciv that incorporates a repair algorithm and a version of Freeciv without a repair algorithm. We observed failures due to looping, segmentation faults, and violated assertions. Figure 9-23 presents the results for the Freeciv benchmark. Without repair, more than half of the injected faults caused Freeciv to crash. With repair, approximately 1.2% of the injected faults caused Freeciv to crash. From these results, we can see that the generated repair algorithm effectively enabled Freeciv to recover from most injected errors.

The crashes we observed in the presence of repair are due to missing consistency properties in our specification. It turns out that our specification is missing some additional consistency properties that constrain the terrain values of tiles. For example, we are missing a constraint that ships must always be in the water. When an ocean tile containing a ship is corrupted and repaired with a land terrain value, an assertion occurs. It is interesting to note that even though the specification is missing consistency properties, the generated repair algorithm is still highly effective in enabling Freeciv to continue to execute.

Note that many of the injected faults do not cause either version to crash. This

can happen because game plays out in a way that Freeciv does not use the terrain value, that the specific terrain value does not cause Freeciv to perform an illegal operation, or the faulty value is a legal terrain value that does not violate any of the consistency properties that Freeciv expects from its data structures.

Version	Number of Successful Runs (out of 10,000)
Repair	9,886
No Repair	4,589

Figure 9-23: Results of Randomized Fault Injection for Freeciv

9.8 Comparison with Our Previous System

We have developed two specifications for CTAS, Freeciv, and the file system: one that requires the developer to provide external consistency constraints to explicitly translate the repaired model to the data structure [14], and the specifications discussed in this dissertation, which use goal-directed reasoning to automatically generate this translation. We found that the new specifications were much simpler to write because goal-directed reasoning eliminated two important potential sources of errors. First, because our new system automatically generated the data structure updates, we did not have to develop a (potentially buggy) set of external consistency constraints to translate the repaired model back into the concrete data structures. Second, and more importantly, our new system eliminated the possibility that the data structure repair algorithm could generate a consistent model with no corresponding data structure. With our old system, the developer had to reason about all of the potential repair sequences to determine if any such sequence might generate such a repaired model. Moreover, the only way to eliminate a repair sequence that produced a repaired model with no corresponding data structure was to develop additional consistency constraints to eliminate problematic repair sequences.

Another advantage is that the new specifications for Freeciv and the Linux file system are approximately 14% smaller than the old specifications. For CTAS, the new

system enabled us to add some additional constraints while maintaining a specification of approximately the same size.

We did not develop an AbiWord specification for the previous system. While it is difficult to estimate exactly how difficult it would have been to develop such a specification, we believe that it would have been much more difficult than developing the specification for our current system. Specifically, we believe that our specification would have allowed repaired models with no corresponding data structures. Moreover, it would have been impossible to develop additional consistency constraints that would have ruled out these repaired models. The only recourse would have been to reason about the potential repair sequences that the system could have performed in an attempt to determine if the repair sequences would actually generate a repaired model with no corresponding data structure.

Specifically, our initial AbiWord specification would likely have had a set containing the fragments in the document and a relation modelling the linking relation between these fragments. If the previous repair algorithm added an object to this set without updating the linking relation (in the old system there was no way to state the correspondence between the set and the relation), it would have generated a consistent model that does not correspond to any data structure. As a result, the previous repair algorithm would fail to generate a consistent data structure.

Goal-directed reasoning eliminates this possibility — because it maintains the connection between the concrete data structure and the abstract model, any addition to the set also updates the relation. Our new system therefore ensures that whenever the algorithm updates the set, it also appropriately updates the relation and the concrete data structures. The net effect is that the developer can simply write the specification and be assured that the repaired data structures will satisfy the consistency properties.

We did not develop a specification for the x86 emulator for our previous system. We believe, however, that our specification for the previous system would not have produced repaired models with no corresponding data structures. The primary benefits for this benchmark are therefore the elimination of the possibility of errors in the

external consistency constraints and a shorter specification.

9.9 Discussion

We found it easy to write the consistency specifications for the data structures that our study focused on without having a deep understanding of the applications. Our results show that if the consistency specification detects an error, data structure repair is likely to enable the application to continue to execute successfully.

Two of our benchmarks, the parallel x86 emulator and the Linux file system, were able to use redundant information to repair the data structures with correct values. The repair routine was able to regenerate the correct values for the size of the cache structure in the x86 emulator and the block and inode bitmaps in the file system benchmark. The remaining benchmarks replace inconsistent values with possibly different values that satisfy the consistency properties. It is interesting to note that these benchmark applications continued to execute even if the repair algorithm repaired the data structure with the "wrong" value provided that this value makes the data structure consistent.

All of specifications were partial. They only contained constraints for specific data structures in the application, and only some of the consistency properties for these parts. Our results show that partial specifications can give significant benefits. Our specification for the Freeciv benchmark was missing invariants that constrain the `terrain` field, yet data structure repair was able to successfully repair inconsistencies in this field.

We observed some scaling issues with the node pruning step. The node pruning step searches for a set of nodes that when removed from the repair dependence graph gives a repair algorithm that can repair any constraint violation and terminates. This search is exponential in the number of nodes it searches. While we were able to generate repair algorithms for all of the specifications, many of the specifications were close to the limit of what our current node pruning algorithm can reasonably handle. However, we believe that improvements to the search strategy and to the

node pruning heuristics would significantly help this problem.

Determining when the consistency specification should be checked is a significant issue in incorporating data structure repair. One approach is to check and possibly repair the consistency properties when a data structure first enters the system. We used this approach for the file system and CTAS benchmarks. A second approach is to perform the checks when a data structure is updated. We used this approach for the piece table in AbiWord and the cache in the parallel x86 emulator. A third approach is to simply periodically performs checks on the data structure. We used this approach in Freeciv. Ideally, it is best to check the consistency properties as close as possible to the original error to prevent the inconsistency from causing the application to produce unacceptable behavior or crashing or propagating further through the data structures.

When we chose the benchmarks for our study, we found that many software errors fell outside the scope of our technique. In these cases, the software system often simply performed the wrong action. This result of these incorrect actions was that the software typically performed a task incorrectly or, in some cases, it crashed. In these cases, data structure repair is not relevant as these errors do not manifest themselves as a data structure inconsistency.

While performing this study, we identified many important questions that were outside the scope of the current study. It is unclear how much effort is required for the developer to write full specifications for an application. To address this issue we are currently working on automatically inferring these specifications. This work uses Daikon [16] to automatically infer data structure invariants, and then translates the Daikon invariants into the data structure consistency specifications used by the specification compiler. This tool can automatically infer many consistency properties, and the developer can augment the inferred specification with additional properties or remove undesired properties.

Furthermore, it is also unclear whether the developer would accidentally leave important properties out of these specifications. It may be possible to test how well the specification covers the consistency properties by randomly injecting faults

into the data structures and investigating the cause of any unacceptable behavior or crashes that occur.

Another issue with full specifications is the overhead of checking such specifications. While the overheads we observed were insignificant, these overheads may be significant if the repair algorithm checked consistency properties for every data structure in the application.

The next step is perform an evaluation in which we first develop a consistency specification for an application, then find errors in the application, and finally test whether the repair algorithm enables the software to recover from these errors. This evaluation is more difficult to perform because it requires understanding the complete code for several applications and then writing a complete specification for each of these application. However, this evaluation more closely mimics how we expect data structure repair to be deployed.

Our experience using data structure repair has helped us discover properties that make applications amenable to this technique. One issue is whether the state in the application are easy to access. It is much easier to write a consistency specification for a few global data structures than to write a consistency specification for thousands of local variables. Furthermore, it is much easier to write a consistency specification if these data structures are either well-specified or easily understood. A related issue is whether the data structures are simple to decode. Our system works well for standard pointer-based heap structures. However, it is difficult to write a specification for our current system that parses information stored as a complex textual string (such as an HTML document). Another issue is that the data structures must have important consistency properties for the technique to be useful. It is possible to write data structures for which nearly all states correspond to legal values. Since such data structures are never inconsistent, data structure repair is not useful. Finally, the application must be important enough that the developer is motivated to write consistency specifications.

Chapter 10

Related Work

We survey related work in software error detection, traditional error recovery, manual data structure repair, and databases.

10.1 Rebooting

Rebooting provides a common approach to error recovery — when a system enters an unresponsive or erroneous state users simply reboot the system. Rebooting is a remarkably effective method to recover from errors provided that the system can tolerate losing all of its volatile state and the downtime necessary to reboot the system. While this is acceptable for some systems, there are many systems that cannot tolerate this loss.

Recurring problematic inputs are a significant problem for the rebooting solution. If a system has a problematic recurring input, rebooting will not be successful, the system will simply crash again when it attempts to reprocess the problematic input. Inconsistencies in persistent state are also a significant problem: rebooting will not re-initialize the persistent state, and therefore when the application processes this state it will crash again. If the application is in the process of updating persistent state when it is rebooted, it may leave this persistent state in an inconsistent state. When it reboots, the inconsistent persistent state may cause the application to produce unacceptable results or even crash.

Recently, this rebooting technique has been extended to enable rebooting a minimal set of components rather than the complete system [9]. This approach divides a computation into many individually rebootable modules. The system reboots a minimal set of modules to recover from an inconsistent state. Note that this approach still suffers from the persistent state problem previously described.

10.2 Checkpointing

Many applications run for long periods of time, and transient errors in such computations can result in losing large amounts of work. In this context, checkpointing serves to avoid the data loss caused by transient errors during a computation [38, 31, 12, 37]. Checkpointing periodically saves the state of the computation, and if the computation crashes, recovery is used to restart the computation from the latest safe point. It has been commonly used in long running scientific computations.

Checkpointing provides correctness guarantees that data structure repair does not. Provided that there are no latent errors stored in the checkpointed state, when an application is restored from a checkpoint, the data structures are completely correct. When an inconsistent data structure is repaired, it may differ from the data structure that a correct execution would have produced.

However, if the checkpointed state is inconsistent due to a latent error, the restored application will run on inconsistent data structures and may produce unacceptable results or even crash. Furthermore, checkpointing is limited in that it cannot handle recurring deterministic errors because recovering from the checkpointed state will simply result in the application crashing repeatedly when the same erroneous code is executed. It is also difficult to perform actions that affect the physical world or that communicate with external systems in checkpointed systems. The problem is that if the system crashes, the restarted application does not know whether these actions were performed after the latest checkpoint. As a result the application may repeat these actions when the system is recovered from a checkpoint. Since data structure repair does not rely on reverting the execution of an application to a previous point,

it enable applications to execute through erroneous computations, support actions that affect the physical world, and communicate with external systems.

10.3 Transactions

Database systems use a combination of logging and replay to avoid the state loss normally associated with rolling back to a previous checkpoint [18]. Transactions support consistent atomic operations by discarding partial updates if the transaction fails before committing. Researchers have recently added hardware support for transactional memory [7]. Transactional memory allows normal applications to incorporate the advantages of transactions. These techniques allow a software system to revert to a completely correct state provided that all erroneous operations were reverted. However, if a software system has performed an operation that cannot be rolled back since the last consistent state, these techniques cannot be used. As a result, it is difficult to build transactional systems that interact with the physical world or external systems. Furthermore, if a transaction has a recurrent error, dependent transactions will not be executed and the original transaction will be aborted. Some systems may loop in this situation, these systems may repeatedly retry the transaction every time the recurrent error causes it to fail. Data structure repair allows the computation to make progress in these situations.

There has recently been renewed interest in applying many of these classical techniques in new computational environments such as Internet services [30]. The ROC project has developed an undoable email system. This system is designed to address latent user errors by enabling the user of the email system to rewind the email system to the point at which a configuration error was made, repair the configuration error, and then replay all of the subsequent operations. This allows the user to retroactively repair configuration errors.

10.4 Manual Data Structure Repair

The Lucent 5ESS telephone switch [21, 20, 24, 19] and IBM MVS operating system [29] use inconsistency detection and repair to recover from software failures. The software in both of these systems contains a set of hand-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 [18].

Fsck [5], chkdsk [3], and scandisk detect and repair inconsistencies in file systems. These applications consist of hand-coded procedures that inspect file systems at boot time to find and repair any inconsistencies. They are primarily targeted at inconsistencies caused by improperly shutting down the computer. These hand-coded applications use domain-specific repairs such as storing the contents of unlinked inodes and blocks in files to enable the user to recover their contents, replicating any blocks that are shared between files, and recovering file system layout information from redundant super blocks. As a result of these domain-specific repair actions, the hand-coded file system repair utilities may preserve more information than our automatically-generated repair algorithms. While our automatically-generated repair algorithms do not currently perform these domain-specific repair actions, some of these repair actions may be general enough that future versions of repair system might include them. Finally, the hand-coded repair algorithms have the potential to be more efficient. Our automatically-generated repair algorithms construct abstract models of the entire data structure. Hand-coded repair algorithms may be able to check the constraints directly on the data structure or to optimize what information is stored in auxiliary data structures. While we have not performed any experiments that compare the performance of our automatically-generated repair algorithms to hand-coded repair algorithms, we expect that developers can more easily optimize hand-coded repair algorithms.

Testing and debugging these hand-coded repair algorithms can be challenging. The developer must reason about all of the possible ways that a file system may be corrupted, and about the effects of repair actions on arbitrarily broken file systems.

This requires that the developers write code that is highly defensive and makes no assumptions about the state of file system. It is not surprising that various implementations of file system repair utilities have contained serious errors [4, 2]. We believe that testing and debugging the specifications used to automatically generate repair algorithms is likely to be easier than testing and debugging hand-coded repair algorithms, and therefore, automatically-generated repair algorithms are less likely to contain errors.

10.5 Constraint Programming

Researchers have incorporated constraint mechanisms into programming languages. One such system is Kaleidoscope [26]. The developer is intended to write applications using a hybrid of imperative style programming and constraints. Kaleidoscope contains a constraint solver, and the imperative application operates by adding and removing constraints to and from the set of constraints to be solved. The developer can use constraints to define a variables in terms of other variables, and then the constraint solver automatically updates that value of the variable whenever the variables it depends on are updated. Programmers are intended to use this functionality to write constraint to relate multiple values, then only update one of these values and rely on the constraint solver to update the remaining values. This constraint solving framework can be viewed as a form of data structure repair in which the user performs an operation and the constraint solver updates the state to satisfy any violated constraints.

Kaleidoscope does not include any analog of our model-based approach, and 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. Furthermore, Kaleidoscope constraints do not include quantifiers. As a result, a developer cannot write constraints that apply to an entire class of objects as he or she could using our data structure repair tool. Kaleidoscope constraints are applied to values within a constraint solver, and not to the arbitrary data structures that our data structure

repair tool supports. Finally, Kaleidoscope does not provide a guarantee that it can satisfy the constraints: if it cannot simultaneously satisfy all constraints, it simply exits with an error.

Kaleidoscope uses a constraint solver for all computation (except branching and adding or removing constraints) in the application. The author notes that "Overall, the programmer still pays a performance penalty for using constraint imperative programming over conventional imperative programming based on these language implementations with exclusively runtime optimizations." Since data structure repair can be performed infrequently, it likely has lower overheads.

Rule-based programming [28, 25] is a related technique in which the developer writes rules with a test condition that, if true, causes an action to take place. The rule-based systems are extensions to imperative programming languages that allow the developer to specify a set of rules. Each rule consists of a condition to be evaluated and an imperative action. If the execution of the application causes the condition to be true, the corresponding action is executed. This mechanism can be used as a framework to enforce constraints, since the conditions can test for constraint violations and the actions can repair these violations. While the rule based system provides a flexible framework for enforcing constraints, it has all of the problems of hand-coded repair routines: the developer must manually develop imperative code to repair constraint violations and then must manually reason that the evaluation of the rules eventually terminates. Our specification compiler automates the generation of the repair code and the reasoning to determine that repairs terminate.

10.6 Integrity Management in Databases

Database researchers have developed integrity management systems that enforce database consistency constraints [11, 36, 10]. 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. Ceri and Widom have developed a system that assists the developer in creating a set of production rules that

maintain the integrity of a database [11]. Database production rules consist of three components: a list of transactions that trigger the rule, a condition to evaluate, and an action to perform if the condition is true. This research automatically generates the list of transactions that trigger a given repair and the condition to be evaluated. The developer has to manually specify the repair action. This research uses a triggering graph to determine whether a set of production rules could potentially trigger each other indefinitely. The triggering analysis is a syntactic analysis that analyzes whether a rule's action performs an update that is mentioned in the list of transactions that trigger another rule. This analysis does not consider the conditions in the production rules. For example, this analysis cannot determine that a production rule that triggers on a given relation, checks some condition on that relation, and potentially performs a repair on that relation may terminate. Our termination analysis is considerably more sophisticated: it reasons about the conditions under which a repair may be performed, the actual repair action, and the predicates in other constraints. For example, our analysis can determine that a repair action that repairs a relation does not trigger itself indefinitely.

The approach of Ceri and Widom has been extended to enable their system to automatically generate both the triggering components and the repair actions for constraints written in standard conjunctive form [10]. This research extends the previously mentioned technique to add the ability to automatically generate repair actions. This work introduces a triggering hypergraph to reason about termination. The triggering hypergraph extends the previous triggering graph to allow the system to eliminate repair actions that may cause non-termination. However, this graph is constructed using the same basic criteria and therefore suffers the same problems. This research introduces the concept of state dependent repairs, repair actions that may fail depending on the initial state. These actions are backed up by state independent repairs, repair actions that are guaranteed to succeed regardless of the original state. While our current repair system only generates state independent repair actions, we could easily augment it to support state dependent repair actions.

Researchers have also developed a database constraint analysis system for Horn

clause constraints and schema constraints (which can constrain a relation to be a function) [36]. This system presents a constraint analysis to determine after a database update what operations must be performed to maintain a set of consistency constraints. Our system supports a broader class of constraints (logical formulas instead of Horn clauses) and automatically generates repair actions for constraint violations.

A key difference between these systems and our system is the form of the data that the repair algorithms operate on. While these systems all repair relational databases, our system repairs the low-level data structures in an application. As a result, our system has to construct an abstract model, translate model repairs into data updates, and reason about this translation. Since these systems repair relational databases, they do not have to address any of these issues. Our system also supports constraints that 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. Our system uses partition information to improve the precision of the termination analysis, enabling the verification of termination for a wider class of constraint systems.

10.7 Data Structure Repair using Search

Khurshid et al. have developed a data structure repair system using search with backtracking [23]. Note that this research was performed after the work described in this dissertation. This system uses Java consistency checking methods as imperative specifications of data structure properties. This system instruments the execution of the consistency checking method to record field accesses. It then uses backtracking search to find field assignments that cause the method to return true. For primitive fields, it uses symbolic execution to construct path conditions that are evaluated using CVC Lite. For object fields, it tries assigning the field to either null, a newly allocated object, or an object that has already been accessed.

One advantage of this approach is that it may be able to handle systems of constraints that our termination analysis rejects. The disadvantages of this approach are that there is no guarantee that the specification is satisfiable, the search may fail

to terminate for certain constraints, it is unclear how well this method can handle low-level memory layout errors, and it is hard to reason about what repairs may be performed.

10.8 Specification-Based Data Structure 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 [14, 15, 13]. The primary disadvantage of this approach in comparison with the approach presented in this dissertation 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. Writing external consistency constraints for linked data structures can be very difficult; if repairs potentially add objects to the linked data structure, the developer has to explicitly write constraints to ensure that these new objects are properly linked. Such constraints could not be expressed in the consistency constraint language. Our current technique effectively recovers these implicit linking constraints from the model definition rules.

A secondary disadvantage is that this system requires the developer to write a set of external consistency constraints. As described in Section 9.8 the external consistency constraints represent an additional specification burden. Furthermore, since the external consistency constraints are only used when a repair is performed, it may be difficult to test that they are correct.

10.9 File Systems

Some journaling or log-structured file systems are always consistent on the disk, eliminating the possibility of file system corruption caused by a system crash [6, 33]. While such file systems eliminate inconsistencies caused by improper system shutdowns, hardware and software errors can still corrupt the file system. As a

result, such file systems still need check and repair utilities. Data structure repair remains valuable even for these systems in that it can enable the system to recover from file system corruption.

10.10 Object Modelling Languages

Our specification language borrows heavily from object modelling languages such as Alloy [22] or UML [32]. In particular, Alloy uses a similar set and relation based abstraction to model systems. However, the requirements that the repair algorithm generation problem places on our specification language have resulted in some significant differences. Data structure repair requires our specification language to establish a direct connection between our abstract model and the data structures. While Alloy has been used to model data structures, Alloy does not include an equivalent to our model definition language or our structure definition language to establish a direct connection between the abstract model and the actual data structures. As a result, one cannot easily use Alloy to express constraints on physical data structures.

In order to facilitate efficient repair algorithms, we have placed limitations on the constraints in our consistency constraint language. In general, the flexibility of the model definition rules enables developer to still express these constraints. For example, our specification language does not support existential quantifiers, while Alloy supports existential quantifiers. As a result, certain properties are easier to state in Alloy. Alloy also includes a transitive closure operation, and as a result it is very easy to express that a data structure is acyclic. While constraints involving transitive closures can be expressed in our system by using the model definition rules to construct relations that contain the transitive closure, this method is less elegant. Finally Alloy supports many operations on sets including set intersection, set union, and set difference. While our consistency constraint language does not include these operations, the model definition rules can be used to construct sets that contain the result of these operations.

Chapter 11

Conclusion

Data structure repair can be an effective technique for enabling software applications to recover from data structure corruption 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 components of our repair system are the specification language, goal-directed reasoning for translating model repairs into data structure updates, and the repair dependence graph for reasoning about the behavior of the generated repair algorithm.

The key elements of our repair technique are as follows:

- **Specification-based Approach:** This specification-based approach to data structure repair allows the developer to use an abstract model to express important data structure consistency properties.

- **Heuristically Close Repaired Data Structures:** Our repair technique generates data structures that are heuristically close the original broken data structure.

- **Automatic Generation of Repair Algorithms:** The specification compiler automatically generates repair algorithms. Our specification compiler contains

several mechanisms that allow the specification developer to influence the repair process. These mechanisms include excluding certain repairs, specify costs associated with repairs, and specifying sources for new objects.

- **Goal-Directed Reasoning:** Goal-directed reasoning analyzes the model definition rules to automatically translate model repairs into data structure updates. This technique frees the developer from specifying how to translate the abstract state to the concrete data structures, while still allowing the developer to utilize the advantages that abstraction provides: a means to classify objects and a means to separate the complexity arising from low-level data structures implementation details from the consistency properties.

- **Repair Dependence Graph:** The specification compiler constructs and analyzes the repair dependence graph to determine whether the generated repair algorithms terminate. This graph captures the dependences between the consistency constraints, the repair actions, and the model definition rules. This graph supports formal reasoning about the effect of repairs on both the abstract model and the data structures. We have identified a set of conditions on the repair dependence graph that when satisfied ensure that all repairs successfully terminate. We have developed an algorithm that removes nodes from the graph to eliminate problematic cycles. These node removals prevent the repair algorithm from choosing repair strategies that may not terminate. Furthermore, the developer can use the repair dependence graph to reason about the behavior of the generated repair algorithms, or if the specification compiler was unable to generate a repair algorithm to understand why the specification compiler failed.

We have evaluated our repair tool using five different applications. Our repair tool enabled these applications to recover from otherwise fatal data structure corruption errors and to continue to execute successfully. Even though this evaluation produced positive results, several questions remain unanswered. How hard is it to obtain specifications that have good coverage of the properties that software errors will violate? How well will data structure repair work on a broader class of applications

with (presumably) a broader class of errors? The next step would be to perform an evaluation in which we first develop a consistency specifications for a broader class of applications, find errors in the applications, and finally test whether the repair algorithm enables the applications to acceptably recover from these errors.

We believe that our specification-based 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 applications.

11.1 Future Work

Our experience with data structure repair suggests many future research directions.

- **Checking Efficiency:** Although our current implementation is able to efficiently check many specifications, certain applications may require more frequent checking of consistency properties or checking larger data structures than our current system can support. Partially checking specifications would enable more frequent consistency checking. Approaches to partial checking include: using hardware support to incrementally check only the changed portions of data structures, using static analysis to eliminate unnecessary checks, and only checking consistency properties for the state that the current method will access.

- **Specification Language Improvements:** Some consistency properties cannot easily be expressed in the current specification language (for example constraints on floating point numbers) and certain data structures are difficult to parse (for example HTML documents). Further experience using the current system will likely suggest ways that the specification language should be improved to more easily parse common data structures and to be able to express more of the important consistency properties for data structures.

  The current specification language requires that the developer abstract any state that a consistency constraint references. For some simple consistency properties, this requirement makes specifications needlessly longer. Allowing

consistency constraints to directly reference fields and array elements would allow the developer to avoid writing unnecessary model definition rules. This would also result in more efficient consistency checking as the abstract models would be smaller.

- **Repair Generation Improvements:** The current system is somewhat limited in the types of repairs that it can generate. For example, for file system blocks that are illegally shared by two files, the specification compiler currently generates repairs that remove the block from one of the two files. A better repair option might be to replicate the block, leave the original in one file, and put the replica in the other file. Further experience examining applications would likely suggest other such repair actions that should be generated by the specification compiler. Furthermore, the current repair actions often rely on the form of a constraint. For example, the current repair algorithm satisfies inequalities by changing the value on the left hand side. Future repair algorithms might generate better repairs by performing repairs on the values appearing in the right hand side. Such a system should also include mechanisms to allow the developer to specify preferences as to how the repair algorithm should repair a given basic proposition.

- **Developer Interface Improvements:** The current specification compiler generates repair algorithms with little input from the developer. As a result, it sometimes selects undesired repair actions. A more interactive system might provide a graphical interface that allows the developer to easily provide guidance for selecting the best repair. Another issue is that the termination analysis is conservative. In some cases, the developer might know that a given repair action will not violate a constraint even though the specification compiler is unable to show this. If the developer provided this information to the specification compiler, the specification compiler may be able to generate better repairs or even generate repair algorithms in which it would otherwise fail to generate a terminating repair algorithm.

- **Specification Coverage:** One open question with this research is how hard is it to obtain specifications of all the data structures in the application. To address this issue we are currently working on automatically inferring these specifications. This work uses Daikon [16] to automatically infer data structure invariants, and then translates the Daikon invariants into the data structure consistency specifications used by the specification compiler. A related issue is that the data structure specifications may under specify the consistency constraints. One possible way to address this issue is to randomly inject faults into the data structures. If data structure repair does not enable the application to recover from one of these errors, the injected fault may have violated one of the unstated consistency constraints. The developer may be able analyze the crash to discover what constraint was missing. It would be interesting to empirically investigate how effective this randomized fault injection technique is at finding unstated invariants.

- **Related Repair Techniques:** Data structure repair can be viewed as a method to prevent the propagation of faults. The original error causes a data structure invariant to be violated, which without repair may further propagate eventually resulting in a crash. Data structure repair prevents the error from propagating through data structure inconsistencies. However, software errors can propagate through other means. For example, the effects of a software error can propagate through application control flow. Developing other techniques that isolate the effects of software errors would likely allow the construction of more robust software applications.

11.2 Implications

Data structure repair may be a first step in changing how we address errors in software. Our current strategy for handling software errors is to try to construct error free software. We try to write software with as few errors as possible. Unfortunately, we make some errors in this process. We attempt to find these errors by using tests that

thoroughly exercise the application. When a test uncovers an error, the developer studies the code to understand the underlying error, and then patches the code to remove this error. Unfortunately, testing does not catch or correct all errors. Errors may not be triggered by the tests, or the developer may introduce new errors with the patch.

The implicit assumption in the current software development methodology is that it is best to attempt to construct perfect software. Unfortunately, one result of this perspective is that developers make little effort to enable software to recover from errors. The problem with this approach is that humans are imperfect, and as a result, the software systems that we construct inevitably have errors. Since we typically do not design software systems to be robust with respect to internal errors, they frequently fail catastrophically when an error occurs.

Data structure repair is part of a fundamentally different approach to handling software errors. This approach suggests that the way to handle errors is to recover and proceed even in the absence of correctness guarantees. The philosophy behind data structure repair acknowledges that software systems contain errors. While we would like to construct completely correct software systems, unfortunately, in practice it is currently impossible to develop correct software systems for any reasonably complex task. Instead, this new approach focuses on developing software systems in such a way that systems can tolerate software errors without causing unacceptable behavior.

If we look at complex systems in other disciplines, we find out that they have both errors and mechanisms to handle their errors. For example, complex biological systems incorporate repair mechanisms to more effectively tolerate failures; small failures in biological systems typically do not result in death. Our bodies include mechanisms to repair both macroscopic and microscopic damage: our bodies can repair cuts, broken bones, DNA transcription errors [8], and damage inflicted by bacteria and viruses.

Our automatically-generated data structure repair algorithms take the first step in bringing repair to software systems. It enables software systems to repair data structure consistency violations. The goal of this line of research is to enable software

systems to survive the damage caused by errors, and to continue to provide useful services.

This technique may provide us with a new way of engineering software systems. We currently design software systems by assuming that all operations have been executed correctly. Unfortunately, for large, long-running systems this assumption is almost certainly false. Since this assumption is deeply ingrained in the software system's design, the system is extremely brittle to any errors and violations of this assumption caused by a single error will likely cause the system to fail catastrophically.

In the longer run, data structure repair may play an important role in making software systems more robust to these kinds of errors. We may be able to build software systems that are designed to extensively incorporate data structure repair and to communicate with the repair system. This would allow us to develop, for example, an air traffic control system that can automatically heal itself from data structure corruption errors and inform the user if any information may be untrustworthy. Future data structure repair systems may bring positive aspects of biological systems to computer systems: robustness to small errors, adaptability to unexpected situations, and an understanding of the software system's health to the user.

Bibliography

[1] Center-tracon automation system. http://www.ctas.arc.nasa.gov/ .

[2] Changelog in reiserfsprogs distribution. ftp://ftp.namesys.com/pub/reiserfsprogs/.

[3] Chkdsk. http://www.microsoft.com/resources/documentation/Windows/XP/all/reskit/en-us/Default.asp?url=/resources/documentation/windows/xp/all/reskit/en-us/prmb_tol_pwfd.asp.

[4] E2fsprogs release notes. http://e2fsprogs.sourceforge.net/e2fsprogs-release.html.

[5] Ext2 fsck manual page. http://e2fsprogs.sourceforge.net/.

[6] Whitepaper: Red Hat's new journaling file system: ext3. http://www.redhat.com/support/wpapers/redhat/ext3/index.html.

[7] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *11th International Symposium on High Performance Computer Architecture (HPCA-11)*, February 2005.

[8] V. A. Bohr, K. Wassermann, and K. H. Kraemer. *DNA Repair Mechanisms (Alfred Benzon Symposium No. 35)*. Copenhagen:Munksgaard, 1993.

[9] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS-VIII*, May 2001.

[10] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems*, 19(3), September 1994.

[11] Stefano Ceri and Jennifer Widom. Deriving production rules for constraint maintenance. In *International Conference on Very Large Data Bases*, pages 566–577, 1990.

[12] K.M. Chandy and C.V. Ramamoorthy. Rollback and recovery strategies. *IEEE Transactions on Computers*, C-21(2):137–146, 1972.

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

[14] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2003.

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

[16] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[17] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, May 2003.

[18] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

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

[20] Neeraj K. Gupta, Lalita Jategaonkar Jagadeesan, Eleftherios E. Koutsofios, and David M. Weiss. Auditdraw: Generating audits the FAST way. In *Proceedings of the 19th International Conference on Software Engineering*, 1997.

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

[22] Daniel Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, Laboratory for Computer Science, Massachusetts Institute of Technology, 2000.

[23] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. Repairing structurally complex data. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN)*, August 2005.

[24] David A. Ladd and J. Christopher Ramming. Two application languages in software production. In *USENIX 1994 Very High Level Languages Symposium Proceedings*, October 1994.

[25] Diane Litman, Peter F. Patel-Schneider, and Anil Mishra. 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.

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

[27] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for testing Java programs. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering*, November 2001.

[28] Anil Mishra, Johannes P. Ros, Anoop Singhal, Gary Weiss, Diane Litman, Peter F. Patel-Schneider, Daniel Dvorak, and James 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.

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

[30] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kcman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, March 15, 2002.

[31] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[32] Rational Inc. The Unified Modeling Language. `http://www.rational.com/uml`.

[33] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[34] Beverly D. Sanford, Kelly Harwood, Sarah Nowlin, Hugh Bergeron, Harold Heinriches, Gary Wells, and Marvin Hart. Center/tracon automation system: Development and evaluation in the field. In *38th Annual Air Traffic Control Association Conference Proceedings*, October 1993.

[35] Michael Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. In *IEEE Micro*, Mar/Apr 2002.

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

[37] John W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[38] Youhui Zhang, Dongsheng Wong, and Weimin Zheng. User-level checkpoint and recovery for LAM/MPI. *ACM SIGOPS Operating Systems Review*, 39(3):72–81, 2005.