Data Structure Repair Using Goal-Directed Reasoning

Brian Demsky and Martin Rinard
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

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

We present a new repair algorithm that uses goal-directed reasoning to automatically translate model repairs into concrete data structure repairs. This new repair algorithm eliminates the possibility of incorrect translations and repaired models with no corresponding representation as concrete data structures.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Reliability

Keywords

Data Structure Repair, Data Structure Invariants

1. INTRODUCTION

Programs usually assume that the data structures that they manipulate are consistent. A software error or some other event may cause the data structures to become inconsistent. Data structure repair is a useful technique that can restore the consistency properties and enable the program to continue to execute successfully. Our previous work [7] introduced a model-based approach in which the developer writes a specification to identify the required data structure consistency properties.

This model-based approach involves two views: a concrete view of the data structures as they are represented in the memory and an abstract view that models the data structures as sets of objects and relations between objects. A set of model definition rules translates the concrete data structures to the sets and relations in the abstract model. The key consistency constraints are expressed using the sets and relations in this model. This approach provides several key benefits: 1) it provides a mechanism for separating objects that play different conceptual roles in a computation into different sets — the developer can then specify different constraints to apply to each of these different sets, 2) the model definition rules provide a clean, simple mechanism to specify a traversal of a data structure, and 3) it provides a means to manage the complexity of data structure consistency properties — the model definition rules encapsulate the data structure representation complexity and the model consistency constraints encapsulate the complexity inherent in the consistency property. There are three challenges in making this approach effective: 1) maintaining a correspondence between the abstract model and the concrete data structures, 2) generating a set of repairs that is sufficient to repair any error, and 3) ensuring that all repairs terminate.

Our previous work [7] performed repairs on the abstract model and relied on a set of user-defined *external consistency constraints* to faithfully translate the repaired model state to the data structures. While this approach automates much of the repair process, the presence of the external consistency constraints has several undesirable properties:

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

Our new algorithm replaces the external consistency constraints with a goal-directed reasoning algorithm on the model

*This research was supported in part by a fellowship from the Fannie and John Hertz Foundation, DARPA Cooperative Agreement FA 8750-04-2-0254, DARPA Contract 33615-00-C-1692, the Singapore-MIT Alliance, and the NSF Grants CCR-0341620, CCR-0325283, and CCR-0086154.

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

definition rules. The new approach has several advantages over the previous approach:

- It eliminates the effort involved in writing the external consistency constraints.
- It eliminates the possibility of errors in the external consistency constraints and guarantees that repairs are correctly translated from the model to the data structures.
- It eliminates the possibility that the repair algorithm may produce a model with no corresponding concrete data structure representation.

1.1 Repair Algorithm Generator

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

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

Our compiler uses a *repair dependence graph* to reason about the termination of the generated repair algorithm. The nodes in this graph represent constraints, repair actions, and changes to the sets and relations in the abstract model. The edges capture dependences between the constraints, repair actions, and the abstract model. The absence of certain kinds of cycles in the graph ensures that all repairs will terminate. In addition to analyzing the graph to determine termination, our compiler uses reasoning and search to remove (subject to graph certain consistency conditions) nodes to eliminate undesirable cycles. These removals further constrain the actions of the generated repair algorithm and ensure that the repair algorithm will never choose a repair strategy that leads to an infinite repair loop.

Figure 1 shows a graphical overview of the repair process. The square boxes in the figure correspond to data structures. The rounded boxes correspond to abstract models. The arrows from the square boxes to the rounded boxes map a data structure to the corresponding abstract model. When invoked, the generated repair algorithm constructs the abstract model and examines it to find any inconsistencies. The arrow labelled "Model Construction" in Figure 1 shows this step. Whenever the repair algorithm discovers an inconsistency, it selects an appropriate model repair action to repair the inconsistency in the model. The arrow

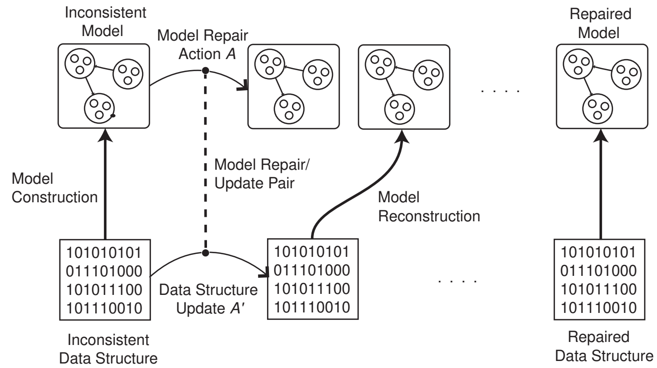

Figure 1: Overview of Repair Process

labelled "Model Repair Action *A*" in the figure shows the model repair action step.

The compiler uses goal-directed reasoning to statically map model repair actions to data structure updates. To implement a model repair that removes an object from a given set, for example, the compiler analyzes the model definition rules to find all the rules whose inclusion constraint may cause the object to be inserted into the set. The compiler then analyzes the guards and the quantifiers of the rules to extract a set of data structure properties whose satisfaction ensures that no rule specifies that the object should be a member of that set. Finally, the compiler generates code to apply (as necessary) a set of data structure updates that force all of these properties to hold. The effect is to remove the object from the set.

After performing the model repair action, the repair algorithm performs the corresponding data structure update. The arrow labelled "Data Structure Update *A'*" in the figure shows this step. Note that there may also be potentially undesirable side effects which cause additional inconsistencies. To ensure that the model reflects these side effects, the repair algorithm must rebuild the abstract model. In Figure 1, the curved arrow labelled "Model Reconstruction" illustrates this model reconstruction. The generated repair algorithm repeats this process to repair all of the inconsistencies.

1.2 Contributions

This paper makes the following contributions:

- **Basic Repair Approach:** It presents an approach that allows the developer to use an abstract model to express important data structure consistency properties. Violations of these properties are repaired by automatically translating model repairs back through the model definition rules to automatically derive a set of data structure updates that implement the repair.
- **Repair Translation:** It presents an algorithm that uses goal-directed reasoning to derive a set of data structure updates that implement the repair.
- **Repair Dependence Graph:** It introduces the repair dependence graph, which captures dependences between consistency constraints, repair actions, and the abstract model. This graph supports formal reasoning about the effect of repairs on both the model and the data structures. It also presents a set of con-

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

Figure 2: Structure Definitions

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

- **Experience:** It presents our experience using data structure repair on several applications. Our experience indicates that data structure repair enables our applications to recover from data structure corruption errors to continue to execute successfully.

2. EXAMPLE

We next present a simple file system example that illustrates the operation of our repair algorithm.

2.1 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 model must satisfy. The translation part for our example consists of the data structure definitions in Figure 2 (these definitions specify the physical layout of the data structures that comprise the file system), the set and relation definitions in Figure 3 (these definitions specify the sets and relations in the model of the file system), and the model definition rules in Figure 4 (these rules specify how to construct the model from the data structures).

We now examine the data structure definitions in Figure 2 in more detail. The `Disk` structure definition specifies that the disk consists of an array of `Block` structures with the `Superblock` being stored in the first block. The superblock defines the layout of the disk. The block bitmap contains a bit for each block in the file system. If the block is used, the bit is set to `true`.

The set declarations in Figure 3 are all 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 relation declarations `relation` $R : S_1$ `->` S_2 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 4 specify how to traverse the data structures to build the sets and relations in the model. Each rule specifies quantifiers that identify the scope of the variables in the body. The body contains a guard and an inclusion condition. The inclusion condition specifies an object (or tuple) that must be in a specific set (or relation) if the guard is true. The least fixed point of the model definition rules applied to the concrete data structure generates the abstract model. The model definition rules for the example are given in Figure 4. The first model definition rule constructs the `SuperBlock` set, and the second constructs the `BlockBitmapBlock` set.

The model constraints in Figure 5 identify the consistency properties that the file system model must satisfy. The first several constraints use the `size` predicate to specify that several sets (`BlockBitmapBlock`, `InodeTableBlock`, `RootDirectoryInode`) must contain exactly one object. The next constraints specify properties that the objects in a given set must satisfy. For example, the 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]. In general, each model constraint is a first-order logical formula consisting of a sequence of quantifiers followed by a quantifier-free boolean formula of basic propositions.

2.2 Repair Algorithm

The generated repair algorithm finds violations of the consistency constraints in the model, synthesizes model repairs that eliminate the consistency violations, then implements these repairs using updates on the concrete data structures. It then repeats the model construction, consistency violation detection, model repair, and data structure repair process until there are no more consistency violations.

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. The second model definition rule in Figure 4 inserts the bitmap block into the set `BlockBitmap`. But because the bitmap block index `d.s.blockbitmapblock` is out of bounds, the rule does not insert a block into the `BlockBitmap` set. The resulting model therefore violates the first constraint from Figure 5. To repair this violation, the repair algorithm transfers a block from the `FreeBlock` set (the developer specifies this set as the source of new `Blocks` to insert into other sets) into the `BlockBitmap` set.

[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`.

```
set Block of Block : UsedBlock | FreeBlock
set UsedBlock of Block : SuperBlock |
  FileDirectoryBlock | InodeTableBlock | 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: Set and Relation Definitions

```
true => d.s in SuperBlock
true => d.b[d.s.blockbitmapblock] as BlockBitmap
  in BlockBitmap
true => d.b[d.s.inodetableblock] as InodeTable
  in InodeTableBlock
for itb in InodeTableBlock, true =>
  itb.itable[d.s.rootdirectoryinode] in
  RootDirectoryInode
for j=0 to d.s.numberofblocks-1,
  !(d.b[j] in UsedBlock) => d.b[j] in FreeBlock
for j=0 to d.s.numberofblocks-1,
  for bbb in BlockBitmap, true =>
  <d.b[j],bbb.blockbitmap[j]> in BlockStatus
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
for e in DirectoryEntry,for itb in InodeTableBlock,
  e.inode!=0 => itb.itable[e.inode] in FileInode
for e in DirectoryEntry,for itb in InodeTableBlock,
  e.inode!=0 => <e, itb.itable[e.inode]> in InodeOf
for j in UsedInode, true =>
  <j,j.referencecount> in ReferenceCount
for i in UsedInode, for j=0 to 11,
  !(i.block[j]=0) => d.b[i.block[j]] in FileBlock
for i in UsedInode, for j=0 to 11,
  !(i.block[j]=0) => <i,d.b[i.block[j]]> in Contents
```

Figure 4: Model Definition Rules

```
size(BlockBitmap)=1
size(InodeTableBlock)=1
size(RootDirectoryInode)=1
for u in UsedBlock, u.BlockStatus=true
for f in FreeBlock, f.BlockStatus=false
for i in UsedInode, i.ReferenceCount=size(InodeOf.i)
for b in FileDirectoryBlock,size(Contents.b)=1
```

Figure 5: Model Constraints

To generate an update that implements this transfer, the compiler analyzes the model definition rule that constructs the `BlockBitmap` set to determine that all objects in this set come from the array `d.b` at offset `d.s.blockbitmapblock`. As a result, the repair algorithm can implement the transfer of the block into the `BlockBitmap` set by calculating the block's index in `d.b` and setting `d.s.blockbitmapblock` equal to this value. To choose an appropriate index, the algorithm 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.blockbitmapblock` to the index `j` of a block `d.b[j]` in the `FreeBlock` set. A side effect of the data structure update is that the block becomes a member of the set `UsedBlock` of used blocks, which removes the block from the `FreeBlock` set.

After this update, the repair algorithm rebuilds the abstract model and discovers several violations of the fourth and fifth constraints in Figure 5. These violations occur because the new bitmap block does not correctly reflect which blocks are free and which blocks are in use. The repair algorithm repairs each of these violations by updating 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 `<u,true>` (and no other tuple with `u` as its first component) is in `BlockStatus`, and similarly for blocks in the set `FreeBlock`.

The translation of the model repairs to the concrete data structures occurs as the model is rebuilt. Whenever the sixth rule in Figure 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.blockbitmap[j]`) in the concrete data structure to the correct value. After all of these updates are performed, the repair algorithm rebuilds the model and finds that all of the model constraints are satisfied. In this case, the repair algorithm has 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 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 program would have produced, especially if the inconsistent data structure contains less information.

2.3 Repair Algorithm Generation

As illustrated in the preceding section, a successful repair algorithm must: 1) traverse the model to find an inconsistency, 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 removes the need for the external consistency constraints, and enables the repair algorithm to guarantee that the repaired data structures satisfy the consistency specification.

A basic issue in repair termination is that repairing one constraint may cause another to be violated. 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). 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.

3. OVERVIEW OF REPAIR ALGORITHM

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

1. **Initial Model Construction:** The repair algorithm initially constructs an abstract model as described in Section 3.1.

2. **Inconsistency Detection:** The repair algorithm evaluates the model 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. If an object (or a tuple) is added to a set (or a relation) or a tuple in a relation is modified, the repair algorithm immediately performs the corresponding data structure update. If an object (or tuple) is to be removed from a set (or a relation), the repair algorithm registers the data structure updates that remove the particular object. Step 6 will perform the corresponding data structure updates as it rebuilds the model. Step 5 performs all other updates.

5. **Data Structure Updates:** The repair algorithm must perform data structure updates to implement the model repair. Section 3.2.2 describes how the compiler uses goal-directed reasoning to generate a set of data structure updates to implement the model repair.

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

3.1 Model Construction

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

3.2 Repairing a Single Constraint

The inconsistency detection algorithm iterates over all values of the quantified variables in the model constraints, evaluating the body of the constraint for each possible combination of the values. If the body evaluates to false, the algorithm has detected a violation and has computed a set of bindings for the quantified variables that make the constraint false. 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 3.

At this point the algorithm has repaired the violated constraint. However, the updates may have violated other constraints. The repair algorithm therefore rebuilds the model and repairs any new or remaining violated constraints.

3.2.1 Model Repair Actions

The model repair action taken to repair a violated basic proposition depends on the form of the proposition. For size propositions, such as `size(BlockBitmap)=1`, the generated repair algorithm simply adds or removes objects (or tuples) from the appropriate set (or relation) to satisfy the proposition. For equality (or inequality) propositions, such as `i.ReferenceCount=size(InodeOf.i)`, the generated repair algorithm calculates a value that makes the proposition true, then assigns the value to the left hand side of the proposition. For inclusion propositions of the form V `in` SE the generated repair algorithm simply adds or removes the specified object (or tuple) to or from the specified set (or relation).

3.2.2 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 to a set (or a tuple to a relation), the compiler finds all model definition rules with an inclusion constraint that may cause the object (or tuple) to be added to the set (or relation). The goal is to synthesize a set of data structure updates that cause the guard of one of these rules to be satisfied, which in turn ensures that the object (or tuple) is in the set (or relation).

We assume the guards are in disjunctive normal form. The compiler chooses a rule, chooses one of the guards' conjunctions, then generates updates to the data structure that ensure that all of the propositions in the conjunction are true. The specific update depends on the form of the proposition; eg. for inequality propositions such as `v.f` $< E$, the update computes E to generate a value that satisfies the proposition, then assigns this value to `v.f`.

The compiler uses a similar strategy to implement repairs that remove an object (or tuple) from a set (or relation). It 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 `struct` field or variable. The compiler generates a data structure update that satisfies the corresponding set of propositions in a dependence-preserving order.

3.2.3 Compensation Updates

Consider a set of concrete data structure updates whose intended effect is to add an object to a set in the abstract model. These updates satisfy the guard of the model definition rule that adds the object to the set. But these updates may also have unintended side effects. For example, they may affect the guards of other model definition rules, which may in turn cause other undesirable changes to the model. It is sometimes possible to generate more precise updates that prevent these changes by performing additional compensation updates that falsify the guards in the model definition rules that caused the additions to take place.

We therefore augment our translation algorithm to analyze the model definition rules to, when possible, automatically generate additional compensation updates to eliminate the undesirable side 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.

3.2.4 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. As illustrated in Section 2, other sets (as specified in the set and relation definition) are one potential source. For primitive types, such as integers, the action can simply synthesize new values. For `structs`, memory allocation primitives are a potential source of new objects. We allow the developer to specify which source to use and, in the absence of such guidance, use heuristics to choose a source.

4. DEVELOPER CONTROL OF REPAIRS

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

The developer can specify that the repair algorithm should not modify certain fields, sets, or relations. The repair algorithm can then provide feedback that characterizes the inconsistencies that can be repaired without modifying these elements. Some repair actions involve adding an object to a set. The developer can specify the source of the object; typical sources are a memory allocator or another set of objects. We similarly allow the developer to control the source of tuples added to relations. Finally, the developer can provide hand-coded routines to repair certain consistency violations.

5. THE REPAIR DEPENDENCE GRAPH

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

5.1 Nodes in Role Dependence Graph

The graph contains the following nodes:

- **Model conjunction nodes:** In disjunctive normal form, each model constraint C_i is of the form $C_i = Q_{i1}, ..., Q_{im} \bigvee_j^{j_{max}} C_{ij}$. There is one node N_{ij} for each conjunction C_{ij} in the model constraint C_i and an additional node $N_{ij'}$, where $j' = j_{max}+l$, for each quantifier Q_{il} in the model constraint.
- **Model repair nodes:** For each basic proposition C_{ijk} in each conjunction C_{ij} there is a set of nodes $\bigcup_l\{A_{ijkl}\}$ corresponding to the model repair actions that the repair algorithm may use to repair that basic proposition. There are also two model repair nodes A_r for each set and relation, one to model insertions, and the other removals.
- **Data structure update nodes:** There is a set of data structure update nodes $\bigcup_m\{R_{ijklm}\}$ for each model repair node A_{ijkl} in the graph. These update nodes represent the concrete data structure updates that implement the repair. There is also a similar set of nodes $\bigcup_s\{R_{rs}\}$ for each model repair node A_r.
- **Increase and decrease scope nodes:** For each model definition rule M_w, there is an increase scope node S_w and a decrease scope node F_w. These nodes represent the side effects that an update has on the model definition rules — in particular, that a data structure update may increase the scope of a model definition rule (i.e., cause the model definition rule to add a new object to a set or a new tuple to a relation) or decrease the scope of a model definition rule (i.e., cause the removal of an object from a set or a tuple from relation).
- **Consequence and compensation nodes:** For each model definition rule M_w, there is a pair of rule consequence nodes $\mathcal{C}_{wT}$ and $\mathcal{C}_{wF}$ 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 may be used to prevent the undesired scope increase of a model definition rule.

5.2 Edges in the Graph

The edges E in the repair dependence graph represent how the model and data structure repairs may affect other parts of the model and data structures. The important dependence chains flow 1) from repaired conjunctions to conjunctions that the repairs may falsify, 2) from repaired conjunctions to quantifiers whose scope the repair may increase or decrease, 3) from data structure updates to conjunctions that the update may falsify, and 4) from data structure updates to quantifiers whose scope the repair may increase or decrease.

For example, there is an edge $\langle N_{ij}, A_{ijkl}\rangle \in E$ from each model conjunction node N_{ij} to each abstract repair node A_{ijkl} that may repair one of the basic propositions in the conjunction. There are other edges to capture dependences between each of the different classes of nodes. These are described in more detail in our technical report [8].

5.2.1 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 a procedure that determines if the repair of a first basic proposition may falsify a second basic proposition (this proposition is taken from the conjunction that the repair of the first proposition may falsify). Our technical report [8] gives the complete set of rules used to determine if the repair of one proposition may falsify a second.

5.2.2 Data Structure and Compensation Updates

Performing an update changes the concrete data structure. This change may cause additional increases or decreases in the scopes of the model definition rules. The repair dependence graph must contain edges from data structure update and compensation update nodes that reflect these changes. The default rule is that updating a field `f` in the concrete data structures may either decrease or increase the scope of any model definition rule that uses `f`, requiring an insertion of a corresponding edge in the repair dependence graph. The algorithm implements exceptions to this rule (and omits the corresponding edges in these cases) for initial additions to a set, updates that effect only a single binding of a model definition rule, and recursive data structures. These exceptions are described in detail in our technical report [8].

5.2.3 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 model 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. Our technical report [8] gives the complete set rules for adding edges.

6. 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. Any infinite repair sequence therefore shows up as a cycle.

The repair dependence graph must be acyclic with the exception of cycles that solely contain scope decrease and consequence nodes, cycles that solely contain scope increase and consequence nodes, or cycles that are not reachable from the model conjunction nodes [2]. The repair algorithm generator may remove model conjunction nodes, data structure update nodes, and consequence/compensation update nodes to satisfy these cyclity constraints. The generated repair algorithm never performs the repair actions that correspond to the deleted nodes. The final graph must satisfy the following conditions in order to ensure that repairs exist for violated constraints: 1) there is at least one model conjunction node for each constraint in the model, 2) each abstract repair node has at least one edge to a data structure update, and 3) each scope increase or decrease node has at least one edge to a consequence or compensation update node.

[2]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 heap, and the actions in unreachable cycles are never used.

7. EXPERIENCE

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

7.1 Methodology

We implemented our data structure repair algorithm. This implementation consists of approximately 20,800 lines of Java code and C code; the implementation compiles specifications into C code that performs the consistency checks and (if necessary) repairs the data structures. 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 a 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. 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.

7.2 AbiWord

AbiWord is a full-featured word processing program available at www.abisource.com. It consists of over 360,000 lines of C++ code, and can import and export many file formats including Microsoft Word documents. It uses a piece table data structure to internally represent documents. The piece table contains a doubly-linked list of the 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 both a section fragment and a paragraph fragment. We developed a specification for the piece table data structure. Our specification consists of 94 lines, of which 70 contain structure definitions. [3]

A bug in version 0.9.5 (and all previous versions) of AbiWord causes AbiWord to attempt to append text to a piece table which lacks a section fragment or a paragraph fragment. This bug is triggered by importing certain valid Microsoft Word documents, causing AbiWord to fail with a segmentation violation when the user attempts to load the

[3]To reduce specification overhead, we developed a structure definition extraction tool that uses debug information in the executable to automatically generate the structure definitions. This tool works for any program that can be compiled with Dwarf-2 debugging information. For AbiWord, we used this tool to automatically generate all of the data structure definitions. The total specification effort for this application therefore consisted of 24 lines of model definition rules and model constraints.

document. We obtained such a document and used our system to enhance AbiWord with data structure repair as described in this paper. Our experimental results show that data structure repair enables AbiWord to successfully open and manipulate the document. Further inspection reveals that loading this document causes AbiWord to attempt to append text to an (inconsistent) empty fragment list. Our repair algorithm detects the attempt to append text to the empty list and repairs the inconsistency by adding a section fragment and a paragraph fragment, breaking any cycles in the fragment list, connecting the fragments using their `next` fields, pointing the `prev` field of each fragment to the previous fragment, and redirecting the `head` pointer to the beginning of the list and the `tail` pointer to the end of the list. The result of this repair is that AbiWord is able to successfully append the text to the list and continue on to read and edit Word documents without the loss of any information. Without repair, AbiWord fails as it attempts to read in the document.

7.3 Parallel x86 emulator

The parallel x86 emulator is a software-based x86 emulator that runs x86 binaries on the MIT RAW machine [15]. The x86 emulator uses a tree data structure to cache translations of the x86 code. To efficiently manage the size of the cache, the emulator maintains a variable that stores the current size of the cache. A bug in the tree insertion method, however, 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, its size is subtracted 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.

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. Our test workload ran gzip on the x86 emulator. Without repair, the emulator stops with a failed assertion. With repair, the emulator successfully executes gzip.

7.4 CTAS

The Center-TRACON Automation System (CTAS) is a set of air-traffic control tools developed at the NASA Ames research center [1]. The system is designed to help air traffic controllers visualize and manage complex air traffic flows. The current source code consists of over 1 million lines of C and C++ code. Versions of this source code are deployed in the continental United States and are in daily use. CTAS maintains data structures that store aircraft data. Our experiments focus on the objects that store the flight plans. These flight plan objects 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.

Our fault insertion methodology attempts to mimic errors in the flight plan processing that produce illegal values in the flight plan data structures. When the program uses these illegal values to access the array of airport data, the array access is out of bounds, which typically leads to the program failing because of an addressing error. 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.

We used a recorded midday radar feed from the Dallas-Ft. Worth center as a workload. 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, there are many classes of errors which rebooting does not solve: the system will often reacquire the data, reprocess it, and fail again for the same reason.

7.5 Freeciv

Freeciv is an interactive, multi-player game available at www.freeciv.org. The Freeciv server maintains a map of the game world. Each tile in this map has a terrain value chosen from a set of legal terrain values. Additionally, cities may be placed on the tiles. Our fault injection strategy changes the terrain values in pseudo-randomly selected tiles 35 times during the execution of the program. 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 containing a city. The specification consists of 191 lines, of which 173 lines contain structure definitions. The principle 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.

Freeciv comes with a built-in test mode in which several automated players play against each other. Our workload simply runs the program in this built-in test mode. The map was configured to contain 4,000 tiles. With repair, the game was able to execute without failing (although the game played out differently because of changed terrain values). Without repair, the game crashed with a segmentation fault caused by indexing an array with an illegal terrain value.

7.6 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. 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. In general, we have found that developing specifications is a straightforward task once one understands the relevant data structures.

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

In all of our tested cases, the algorithm is able to repair the file system and the workload correctly runs to completion. Without repair, files end up sharing inodes and disk blocks and 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.

7.7 Discussion

At this point 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 [7], and the specifications discussed in this paper, 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 say in hindsight 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.

7.8 Performance

To evaluate the performance of our consistency check and repair algorithm, we computed two numbers: 1) the mean time required to perform a consistency check for a consistent data structure, and 2) the mean time required to perform the consistency check and the repair for an inconsistent data structure (rendered inconsistent via fault injection). Table 1 presents the mean consistency check times (over ten trials) for the different applications and the mean consistency check and repair times. In general, the check and repair times are dominated by the model construction overhead. The check and repair times therefore correlate with the number

Application	Time to check consistency (ms)	Time to check and repair (ms)
AbiWord	0.06	0.55
CTAS	0.07	0.15
Freeciv	3.62	15.66
File system	4.22	263.14

Table 1: Time to check consistency and perform repairs

of times the repair process rebuilds the model. For AbiWord the mean number of times that the repair algorithm rebuilds the model is 10, for CTAS the mean is 3, for the file system the mean is 119.2, while for Freeciv the mean is 4.6. The number of times the model is rebuilt is, in turn, correlated with the number of data structure updates that the repair algorithm performs. The mean number of updates is 7 for AbiWord, 1 for CTAS, 59.1 for the file system, and 1.8 for Freeciv. As these numbers show, the fault injection strategy for the file system produces faults that require substantially more data structure updates to repair.

8. RELATED WORK

We survey related work in software error detection [12, 20], traditional error recovery, manual data structure repair, and databases.

Reboot potentially augmented with checkpointing is one approach to error recovery. Database systems use a combination of logging and replay to avoid the state loss normally associated with rolling back to a previous checkpoint [10]. There has recently been renewed interest in applying many of these classical techniques in new computational environments such as Internet services [5] and in extending these techniques to reboot a minimal set of components rather than the complete system [3].

The Lucent 5ESS telephone switch [9, 17, 13, 21] and IBM MVS operating system [16] use inconsistency detection and repair to recover from software failures. Both of these systems contain a set of manually coded procedures that periodically inspect their data structures to find and repair inconsistencies. The reported results indicate an order of magnitude increase in the reliability of the system [10].

Researchers have incorporated constraint mechanisms into programming languages. In Kaleidoscope [14] the developer writes programs using a hybrid of imperative-style programming and constraints. Kaleidoscope does not include an analog of our model-based approach, as a result, it can be very difficult, if not impossible, to express constraints on recursive data structures or other multi-element heap structures. Another example of a constraint maintenance system as a programming abstraction is Alphonse [11]. Rule based programming [2, 6] is a technique in which the developer defines a condition and an action to take in response.

Database researchers have developed integrity management systems that enforce database consistency constraints. These systems typically operate at the level of the tuples and relations in the database, not the low-level data structures that implement this abstraction [4, 19, 22].

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 [18]. Repair remains valuable even for these systems in that it can enable the system to recover from file system corruption caused by other sources such as software errors or hardware damage.

9. CONCLUSION

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

9.1 Acknowledgments

We would like to thank David Wentzlaff for his help with the MIT RAW x86 emulator.

10. REFERENCES

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

[2] A. Mishra et al. R++: Using rules in object-oriented designs. In *OOPSLA*, July 1996.

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

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

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

[6] D. Litman et al. Modeling dynamic collections of interdependent objects using path-based rules. In *OOPSLA*, October 1997.

[7] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, October 2003.

[8] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. Technical Report 950, MIT Computer Science and Artificial Intelligence Laboratory, 2004.

[9] G. Haugk. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.

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

[11] R. Hoover. Incremental computation as a programming abstraction. In *PLDI*, 1992.

[12] J. Corbett et al. Bandera : Extracting finite-state models from Java source code. In *ICSE*, 2000.

[13] D. A. Ladd and J. C. Ramming. Two application languages in software production. In *VHLL*, October 1994.

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

[15] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. In *IEEE Micro*, Mar/Apr 2002.

[16] S. Mourad and D. Andrews. On the reliability of the IBM MVS/XA operating system. *TSE*, September 1987.

[17] N. Gupta et al. Auditdraw: Generating audits the FAST way. In *ISCE*, 1997.

[18] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *SOSP*, Oct. 1991.

[19] S. Ceri et al. Automatic generation of production rules for integrity maintenance. *TODS*, 19(3), September 1994.

[20] S. Hallem et al. A system and language for building system-specific, static analyses. In *PLDI*, 2002.

[21] T. Griffin et al. Generating update constraints from PRL5.0 specifications. In *Preliminary report presented at AT&T Database Day*, September 1992.

[22] S. D. Urban and L. M. Delcambre. Constraint analysis: A design process for specifying operations on objects. *TKDE*, 2(4), December 1990.