Automatic Detection and Repair of Errors in Data Structures

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

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

ABSTRACT

We present a system that accepts a specification of key data structure constraints, then dynamically detects and repairs violations of these constraints. 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 system can effectively repair errors to deliver consistent data structures that allow the program to continue to operate successfully within its designed operating envelope.

1. INTRODUCTION

To correctly represent the information that a program manipulates, its data structures must satisfy key consistency constraints. If a software error or some other anomaly causes the data structures to become inconsistent, the basic assumptions under which the software was developed no longer hold. In this case, the software typically behaves in an unpredictable manner and may even fail catastrophically.

This paper presents a new approach for attacking the data structure inconsistency problem. Instead of attempting to increase the reliability of the code that manipulates the data structures, our system accepts a specification of key data structure consistency constraints. It then dynamically detects and repairs data structures that violate these constraints. Our goal is not necessarily to restore the data structures to the state in which a hypothetical correct program would have left them (although in some cases our system may do this). Our goal is instead to deliver repaired data structures that satisfy the basic consistency assumptions of the program, enabling the program to continue to operate successfully within its designed operating envelope. We have identified two kinds of data structures that are especially appropriate for this approach: 1) long-lived, persistent data structures (such as file systems, application data files, or serialized data structures), and 2) data structures for critical systems in which continued operation even in the face of errors is a paramount concern.

1.1 Basic Technical Approach

Our approach involves two data structure views: a concrete view at the level of the bits in memory and an abstract view at the level of relations between abstract objects. The abstract view facilitates both the specification of higher level data structure constraints (especially constraints of linked data structures) and the reasoning required to repair any inconsistencies.

Each specification contains a set of model definition rules and a set of consistency constraints. Given these rules and constraints, our tool automatically generates algorithms that build the model, inspect the model and the data structures to find violations of the constraints, and repair any such violations. The repair algorithm operates as follows:

- **Inconsistency Detection:** It evaluates the constraints in the context of the current data structures to find consistency violations.
- **Disjunctive Normal Form:** It converts each violated constraint into disjunctive normal form; i.e., a disjunction of conjunctions of basic propositions. Each basic proposition has a repair action that will make the proposition true. For the constraint to hold, all of the basic propositions in at least one of the conjunctions must hold.
- **Repair:** The algorithm repeatedly selects a violated constraint, chooses one of the conjunctions in that constraint's normal form, then applies repair actions to all of the basic propositions in that conjunction that are false. A repair cost heuristic biases the system toward choosing the repairs that perturb the existing data structures the least.

Note that the repair actions for one constraint may cause another constraint to become violated. To ensure that the repair process terminates, we preanalyze the set of constraints to ensure the absence of cyclic repair chains that might result in infinite repair loops. If a specification contains cyclic repair chains, the tool attempts to prune conjunctions to eliminate the cycles.

1.2 Experience

We have used our tool to repair inconsistencies in three applications: a simplified Linux file system, an interactive game, and Microsoft Word files. In this context, we have used our tool to repair bitmaps identifying free and allocated disk blocks, correct reference counts, eliminate inappropriate sharing in linked data structures, correct illegal values stored in arrays, resolve inconsistencies in correlated values stored in different data structures, and ensure that recorded data structure sizes match the size of the corresponding actual data structure. In addition to these repairs, our tool is also able to correct out of bounds pointers in linked data structures, repair incomplete data structures by allocating and linking in new structures, repair back links (such as parent pointers in trees) in linked data structures, and enforce inequality constraints between multiple values.

We found that the specifications for our applications were relatively straightforward to develop once we understood the underlying data structures and that the automatically generated repair algorithms were able to produce data structures that enabled the corresponding programs to continue to operate successfully. In the absence of this repair, the programs usually failed. Our results therefore indicate that our technique may significantly enhance the ability of applications to recover from data structure errors.

1.3 Contributions

This paper makes the following contributions:

- **Specification-Based Approach:** It introduces the concept of using specifications for the automatic detection and repair of inconsistent data structures. It also introduces the concept of using an abstract model of the data structures to facilitate specification development and reasoning in the repair algorithm.
- **Inconsistency Detection and Repair System:** It presents an implemented system and algorithms that, given a specification, automatically detect and repair violations of the specification.
- **Experience:** It presents our experience using our tool for several applications. This experience indicates that it is relatively straightforward to develop the consistency conditions and that the use of our tool enhanced the ability of the applications to continue to operate in the face of errors.

2. EXAMPLE

We next present a simple file system example that illustrates how our tool works. The file system consists of three parts: the directory, a file allocation table (FAT), and an array of file blocks. Each file consists of a linked chain of file blocks. The FAT is a fixed-size array of file block indices that implements the linking structure; specifically, if a block j is in the chain of blocks for a given file, then FAT[j] is the index of the next block in the chain. The FAT may also contain two special values: if FAT[j] $= -1$, then block j is the last file block in its chain; if FAT[j] $= -2$, then block j is not in any chain and is free for allocation. The directory consists of a fixed number of entries. Each entry contains a file name, a flag indicating whether the entry is in use or not, a field indicating the length of the file, and the index of the first block in the file's chain of blocks. Figure 1 graphically presents an (inconsistent) file system with two directory entries and four file blocks. The file system has two files named abst and intro; abst has length 7 and starts at file block 0; intro has length 9 and starts at file block 2.

Even a file system this simple has many consistency constraints. Our implemented system supports a full range of constraints that involve all of the parts of the file system. In this section, we focus on the following FAT constraints:

1. **Chain Disjointness:** Each block should be in at most one chain.
2. **Free Block Consistency:** No chain should contain a block marked as free in the FAT.

Note that these constraints are stated in terms of conceptual entities such as chains of file blocks rather than directly in terms of the concrete bits on the disk. To support the expression of these kinds of constraints at an appropriate level of abstraction, our approach allows the developer to specify a translation from the concrete data structure representation into an abstract model based on relations between abstract objects. The developer can then use this model to state some of the desired consistency constraints.

2.1 Model Construction

Figure 3 presents the object and relation declarations in the model for our example. There are three sets of objects: `blocks`, `used`, and `free`. Together, `used` and `free` partition the set of block indices `blocks`, which is in turn a subset of the set of `integer` objects. The `next` relation models chains of `used` file blocks.

```
set blocks of integer : partition used | free;
relation next: used -> used;
```

Figure 3: Object and Relation Declarations

Figure 4 presents the structure declarations and rules that define the model. The `Disk struct` declaration identifies the disk as consisting of an array of directory entries followed by the FAT array, and then the file blocks. In our example, `NumEntries`, `NumBlocks`, `Length` and `BlockSize` are all constants, but we support more advanced declarations in which such quantities could be stored in data structure fields.

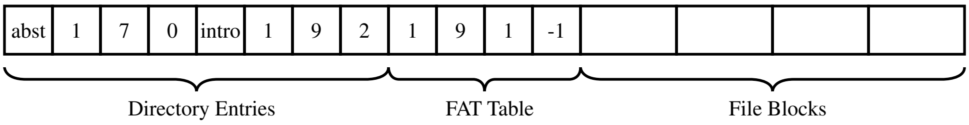
```
struct Entry {
  byte name[Length];
  byte inUse;
  int  size;
  int  firstBlock;
}
struct Block { data byte[BlockSize]; }
struct Disk {
  Entry table[NumEntries];
  int FAT[NumBlocks]; Block
  block[NumBlocks];
}
Disk disk;

for i in 0..NumEntries, disk.table[i].inUse &&
  disk.table[i].block < NumBlocks =>
  disk.table[i].block in used;
for b in used, 0 <= disk.FAT[b] &&
  disk.FAT[b] < NumBlocks => disk.FAT[b] in used;
for b in used, 0 <= disk.FAT[b] &&
  disk.FAT[b] < NumBlocks =>
  <b,disk.FAT[b]> in next;
for b in 0..NumBlocks, !(b in used) => b in free;
```

Figure 4: Model Definition Declarations and Rules

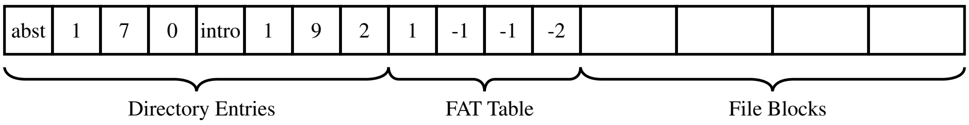
The second part of Figure 4 presents the model definition rules. Each rule consists of a quantifier that identifies the scope of the rule, a guard whose predicate must be true for the rule to apply, and an inclusion constraint that specifies either an object that must be in a given set or a tuple that must be in a given relation. Our tool processes these rules to produce an algorithm that, starting from the directory entries, uses the FAT table to trace out the `next` relation and compute the sets of `used` and `free` blocks.

Note that the rules in Figure 4 use the variable `disk` to refer to the disk image. For long-lived data structures contained in disk images or files, such variables are offsets within



abst	1	7	0	intro	1	9	2	1	9	1	-1			

Figure 1: Inconsistent File System



abst	1	7	0	intro	1	9	2	1	-1	-1	-2			

Figure 2: Repaired File System

the disk image or file. These offsets are defined in a configuration file that we omit here for brevity. For in-memory data structures, the rules use the program variables to refer to the concrete data structures.

2.2 Consistency Constraints

Internal constraints are stated using the model exclusively and not the concrete data structures. Figure 5 presents the internal constraint in our example. This constraint states that each `used` block participates in at most one incoming `next` relation. Note that we use the notation `next.b` to indicate `b` under the inverse of the `next` relation; i.e., the set of all `i` such that $\langle \mathtt{i}, \mathtt{b} \rangle$ in `next`.

```
for b in used, size(next.b) <= 1;
```

Figure 5: Internal Consistency Constraint

The model for the example file system in Figure 1 has the following sets and relations: `used` $= \{0, 1, 2\}$, `free` $= \{3\}$, and `next` $= \{\langle 0, 1\rangle, \langle 2, 1\rangle\}$. File block 1 is in two chains; this inconsistency shows up as a violation of the constraint that `size(next.1)<=1`. To repair this inconsistency, our tool will remove one of the tuples in the `next` relation.

External constraints may reference both the model and the concrete data structures. Figure 6 presents the external constraints in our example. These constraints capture the requirements that the sets and relations in the model place on the values in the concrete data structures. Our tool uses these constraints to implement the model repairs in the concrete data structures. The constraints may also deal with basic representation constraints such as, in our example, the requirement that FAT entries either be -1, -2, or contain a valid file block index. Repairs that enforce these constraints may therefore clean up corrupted values in the data structures.

```
for b in free, disk.FAT[b] = -2
for <i,j> in next, disk.FAT[i] = j;
for b in used, size(b.next) = 0 => disk.FAT[b] = -1
```

Figure 6: External Consistency Constraints

2.3 Repaired File System

Figure 2 presents the repaired file system from Figure 1. In this example, the repair algorithm has chosen to remove $\langle 2, 1\rangle$ from the `next` relation. This repair has eliminated the sharing of file block 1 and truncated the intro file at disk block 2. [1] The repair shows up in the file system as a change in the FAT entry for block 2 from 1 to -1. The repair algorithm has also cleaned up some corrupted values in the FAT table; specifically, it has changed the FAT entry for block 1 from 5 to -1 (indicating that block 1 is the last block in its file block chain) and changed the FAT entry for block 3 from -1 to -2 (indicating that block 3 is free).

3. SPECIFICATION LANGUAGE

Our specification language consists of several sublanguages: a structure definition language, a model definition language, and the languages for internal and external constraints. As described earlier, the structure definition language is similar to that of C. However, it supports a wider range of primitive data types, provides a form of structure inheritance, and allows the developer to define inline, variable-length arrays.

3.1 Model Definition Language

The model definition language allows the developer to declare the sets and relations in the model and to specify the rules that define the model. A set declaration of the form `set S of T partition` $\mathtt{S}_1$`, ...,` $\mathtt{S}_n$ declares a set `S` that contains objects of type `T`, where `T` is either a primitive type (with the range optionally constrained to be between two given values) or a `struct` type declared in the structure definition part of the specification. The set `S` has n subsets $S_1, ..., S_n$ which together partition S. Changing the `partition` keyword to `subsets` removes the requirement that the subsets $S_1, ..., S_n$ partition S but otherwise leaves the meaning of the declaration unchanged. A relation declaration of the form `relation R:` $\mathtt{S}_1 - > \mathtt{S}_2$ specifies a relation between the objects in the sets S_1 to S_2.

The model definition rules define a translation from the concrete data structures into an abstract model. Each rule has a 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 7 presents the grammar for the model definition rules.

Figure 8 gives the denotational semantics $\mathcal{R}[C]\ h\ l\ m$ of a single rule C. A model m is a mapping from set names and relation names to the corresponding sets of objects or relations between objects. 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, $\mathcal{R}[C]\ h\ l\ m$ is the new model after applying the rule to m in the context of

[1] This truncation may leave the length of the file longer than one block. Some (but not all) file systems assume that the length must reflect the number of blocks in the file. If required, it is possible to augment our specification to appropriately constrain the length of the file.

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

Figure 7: Model Definition Language

h and l. Note that l provides the values of both the program variables that the rules use to reference the concrete data structures and the variables bound in the quantifiers.

Each model definition contains a set of model definition rules $C_1, ..., C_n$. Given a model containing these rules, a set of concrete data structures h, and a naming environment l for the program variables , the model is the least fixed point of the functional $\lambda m.(\mathcal{R}[C_1]\ h\ l) \ldots (\mathcal{R}[C_n]\ h\ l\ m)$. The presence of negation in the model definition language complicates the computation of this fixed point. For example, negation makes it possible for a rule to specify that an object is in a given set only if another object is not in another set. We address this complication by requiring the set of model definition rules to have no cycles that go through rules with negated inclusion constraints in their guards.

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

3.2 Pointers

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

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

We next extend the model construction software to check that each `struct` accessed via a pointer is *valid* before it inserts the `struct` into a set or a relation. All valid `structs` reside completely in allocated memory. In addition, if two `structs` 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 `structs` appear in the model.

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

$$
\begin{array}{l}
\mathcal{R}[V\ \texttt{in}\ S, C]\ h\ l\ m\ = \bigcup_{v \in \mathcal{S}[S]\ l\ e} \mathcal{R}[C]\ h\ l[V \mapsto v]\ m \\
\mathcal{R}[\langle V_1, V_2 \rangle\ \texttt{in}\ R, C]\ h\ l\ m\ = \bigcup_{\langle v_1, v_2 \rangle \in S[S]\ l\ e} \\
\quad \mathcal{R}[C]\ h\ l[V_1 \mapsto v_1][V_2 \mapsto v_2]\ m \\
\mathcal{R}[V = E_1\ ..\ E_2, C]\ h\ l\ m\ = \bigcup_{i=\mathcal{E}[E_1]\ h\ l\ m}^{\mathcal{E}[E_2]\ h\ l\ m} \mathcal{R}[C]\ h\ l[V \mapsto i]\ 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\ =!(\mathcal{G}[G]\ h\ l\ m) \\
\mathcal{G}[E_1 = E_2]\ h\ l\ m = (\mathcal{E}[E_1]\ h\ l\ m) == (\mathcal{E}[E_2]\ h\ l\ m) \\
\mathcal{G}[E_1 < 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 = \mathit{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}[E\ \texttt{in}\ S]\ h\ l\ m\ = m \cup \langle S, \mathcal{E}[E]\ h\ l\ m \rangle \\
\mathcal{I}[\langle E_1, E_2 \rangle\ \texttt{in}\ R]\ h\ l\ m\ = m \cup \langle R, \langle \mathcal{E}[E_1]\ h\ l\ m, \mathcal{E}[E_2]\ h\ l\ m \rangle \rangle \\
\mathcal{E}[V]\ h\ l\ m\ = l(V) \\
\mathcal{E}[\mathit{number}]\ h\ l\ m\ = \mathit{number} \\
\mathcal{E}[E.\mathit{field}]\ h\ l\ m\ = b.\langle (\mathcal{E}[E]\ h\ l\ m), \mathit{field}, b \rangle \in h \\
\mathcal{E}[E_1.\mathit{field}[E_2]]\ h\ l\ m\ = \\
\quad c.\langle (\mathcal{E}[E_1]\ h\ l\ m), \mathit{field}, (\mathcal{E}[E_2]\ h\ l\ m), c \rangle \in h \\
\mathcal{E}[E_1 \oplus E_2]\ h\ l\ m\ = \mathit{primop}(\oplus, (\mathcal{E}[E_1]\ h\ l\ m), (\mathcal{E}[E_2]\ h\ l\ m)) \\
\mathcal{E}[\mathit{string}]\ h\ l\ m\ = \mathit{string} \\
\mathcal{S}[S]\ l\ m\ = \{ s \mid s \in m(S) \}
\end{array}
$$

Figure 8: Denotational Semantics for the Model Definition Language

A final complication is that expressions of the form $E.\texttt{f}.\texttt{g}$ may appear in guards. If $E.\texttt{f}$ is not valid, $E.\texttt{f}.\texttt{g}$ is considered to be undefined. Expressions involving undefined values also have undefined values. Comparison ($E_1 < E_2$, $E_1 = E_2$) and set inclusion (E `in` S, $\langle E_1, S_2 \rangle$ `in` R) predicates involving undefined values have the special value `maybe`. We use three-valued logic to evaluate guards involving `maybe`.

3.3 Internal Constraints

Figure 9 presents the grammar for the internal constraint language. Each constraint consists of a sequence of quantifiers $Q_1, ..., Q_n$ followed by body B. The body uses logical connectives (and, or, not) to combine basic propositions P.

Figure 10 provides the denotational semantics for this language. Given a constraint C and a model m, $\mathcal{EV}[C]\ \emptyset\ m$ is `true` if the constraint is satisfied in m and `false` otherwise. The primary complication in the semantics has to do with arithmetic and logical expressions involving relations. Consider, for example, an expression of the form $V_1.R_1 + V_2.R_2$. Strictly speaking, $V_1.R_1$ is the set of objects in the image of

$$
\begin{array}{rcl}
C & := & Q, C \mid B; \\
Q & := & \texttt{for}\ V\ \texttt{in}\ S \mid \texttt{for}\ V = E\ ..\ E; \\
B & := & B\ \texttt{and}\ B \mid B\ \texttt{or}\ B \mid !B \mid (B) \mid P; \\
P & := & VE = E \mid VE < E \mid VE <= E \mid VE > E \mid \\
 & & VE >= E \mid V\ \texttt{in}\ SE \mid \texttt{size}(SE) = 1 \mid \\
 & & \texttt{size}(SE) >= 1 \mid \texttt{size}(SE) <= 1; \\
VE & := & V.R; \\
E & := & V \mid E - E \mid number \mid string \mid E + E \mid E/E \mid \\
 & & E * E \mid E.R \mid \texttt{size}(SE) \mid (E); \\
SE & := & S \mid V.R \mid R.V;
\end{array}
$$

Figure 9: Internal Constraint Language

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

$$
\begin{array}{l}
\mathcal{EV}[\texttt{for}\ V\ \texttt{in}\ SET, C]\ l\ m\ = \\
\quad \wedge\{\mathcal{EV}[C]\ l[V \mapsto v]\ m \mid v \in \mathcal{S}[S]\ l\ m\} \\
\mathcal{EV}[B]\ l\ m = \mathcal{C}[B]\ l\ m \\
\mathcal{C}[!B]\ l\ m = !\mathcal{C}[B]\ l\ m \\
\mathcal{C}[B_1\ \texttt{and}\ B_2]\ l\ m = \mathcal{C}[B_1]\ l\ m \wedge \mathcal{C}[B_2]\ l\ m \\
\mathcal{C}[B_1\ \texttt{or}\ B_2]\ l\ m\ = \mathcal{C}[B_1]\ l\ m \vee \mathcal{C}[B_2]\ l\ m \\
\mathcal{C}[P]\ l\ m\ = \mathcal{PR}[P]\ l\ m \\
\mathcal{PR}[V\ \texttt{in}\ SE]\ l\ m\ = l(a) \in \mathcal{SE}[SE]\ l\ m \\
\mathcal{PR}[VE = E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m == \mathcal{E}[E]\ l\ m) \\
\mathcal{PR}[VE < E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m < \mathcal{E}[E]\ l\ m) \\
\mathcal{PR}[VE \leq E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m \leq \mathcal{E}[E]\ l\ m) \\
\mathcal{PR}[VE > E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m > \mathcal{E}[E]\ l\ m) \\
\mathcal{PR}[VE \geq E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m \geq \mathcal{E}[E]\ l\ m) \\
\mathcal{PR}[\texttt{size}(SE) = 1]\ l\ m\ = \mathcal{E}[\texttt{size}(SE)]\ l\ m\ == 1 \\
\mathcal{PR}[\texttt{size}(SE) >= 1]\ l\ m\ = \mathcal{E}[\texttt{size}(SE)]\ l\ m\ \geq 1 \\
\mathcal{PR}[\texttt{size}(SE) <= 1]\ l\ m\ = \mathcal{E}[\texttt{size}(SE)]\ l\ m\ \leq 1 \\
\mathcal{V}[V.R]\ l\ m\ = y.\langle l(V), y\rangle \in m(R) \\
\mathcal{S}[S]\ l\ m\ = \{s \mid s \in m(S)\} \\
\mathcal{E}[\texttt{size}(SE)]\ l\ m\ = \mid \mathcal{SE}[SE]\ l\ m \mid \\
\mathcal{E}[V]\ l\ m\ = l(V) \\
\mathcal{E}[E.R]\ l\ m\ = y.\exists z, z \in \mathcal{E}[E]\ l\ m \wedge \langle z, y\rangle \in m(R) \\
\mathcal{E}[E_1 \oplus E_2]\ l\ m\ = primop(\oplus, \mathcal{E}[E_1]\ l\ m, \mathcal{E}[E_2]\ l\ m) \\
\mathcal{SE}[S]\ l\ m\ = \{s \mid s \in m(S)\} \\
\mathcal{SE}[V.R]\ l\ m\ = \{y \mid \langle l(V), y\rangle \in R\} \\
\mathcal{SE}[R.V]\ l\ m\ = \{y \mid \langle y, l(V)\rangle \in R\}
\end{array}
$$

Figure 10: Denotational Semantics for Internal Constraints

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. Our primitive arithmetic and logical operations are designed to take as input two singleton sets and produce the appropriate singleton set. If given a non-singleton set as input, the primitives produce the undefined value. We treat undefined values in this semantics the same way as we do in Section 3.2: we appropriately extend arithmetic operations to work with undefined values and logical operations to work with `maybe` according to the laws of three-valued logic.

3.4 External Constraint Language

Figure 11 presents the grammar for the external constraint language. Each constraint has a quantifier that identifies the scope of the rule, a guard G that must be true for the constraint to apply, and a condition C that specifies either a program variable, a field in a structure, or an array element that must have a given value. Figure 12 provides the denotational semantics for this language. Given a constraint R, a heap h, a naming environment l, and a model m, $\mathcal{R}[R]\ h\ l\ m$ is `true` if the constraint is satisfied for h, l, and m.

4. ERROR DETECTION AND REPAIR

The repair algorithm updates the model and the concrete data structures so that all of the internal and external constraints are satisfied. The repair is organized around a set of repair actions that update the model and/or the data structures to coerce propositions to be true. The algorithm has two phases: during the internal phase, it updates the model so that it satisfies all of the internal constraints. During the external phase, it updates the data structures to satisfy all of the external constraints.

4.1 Error Detection in Internal Phase

The algorithm detects violations of the internal constraints by evaluating the constraints in the context of the model. This evaluation iterates over all values of the quantified variables, 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.

4.2 Error Repair in Internal Phase

The repair algorithm is given a body and variable bindings that falsify the body. The goal is to change the model to make the body true. The algorithm first converts the body to disjunctive normal form, so that it consists of a disjunction of conjunctions of basic propositions. Each basic proposition has a repair action that the algorithm can use to modify the model so that the proposition becomes true. The repair algorithm chooses one of the conjunctions and applies repair actions to its basic propositions until the conjunction becomes true and the constraint is satisfied for that set of variable bindings.

There are three kinds of basic propositions in the internal constraint language: size propositions, inequality propositions, and inclusion propositions. Each proposition can occur with or without negation; the actions repair the propositions as follows:

4.2.1 Size Propositions

Size propositions are of the form $\texttt{size}(S) = 1$, $!\texttt{size}(S) = 1$, $\texttt{size}(S) >= 1$, or $\texttt{size}(S) <= 1$ where S can be one of the sets in the model or a relation expression of the form $R.v$ or $v.R$. It is straightforward to generalize size propositions to involve arbitrary constant sizes.

If S is a set in the model, the repair action simply adds or removes items to satisfy the constraint. The action ensures that these changes respect any `partition` constraints between sets in the model. Note that this basic approach also works for negated size propositions. If S is a relation expression, the repair action adds or removes tuples from the relation to satisfy the constraint.

$$\begin{array}{rcl}
R & := & Q, R \mid G \Rightarrow C \\
Q & := & \texttt{for}\ V\ \texttt{in}\ S \mid \texttt{for}\ \langle V, V\rangle\ \texttt{in}\ R \mid \texttt{for}\ V = E\ ..\ E \\
G & := & G\ \texttt{and}\ G \mid G\ \texttt{or}\ G \mid !G \mid E = E \mid E < E \mid true \\
C & := & HE.field = E \mid HE.field[E] = E \mid V = E \\
HE & := & V \mid HE.field \mid HE.field[E] \\
E & := & V \mid number \mid string \mid E.R \mid E - E \mid E + E \mid \\
 & & E * E \mid E/E \mid \texttt{size}(SE) \mid \texttt{element}\ E\ \texttt{of}\ SE \\
SE & := & S \mid V.R \mid R.V
\end{array}$$

Figure 11: External Constraint Language

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

$$\begin{array}{l}
\mathcal{R}[\texttt{for}\ V\ \texttt{in}\ S, R] h_0\ l\ m\ = h_n . h_j = \mathcal{R}[R] h_{j-1} \\
\quad l[V \mapsto v_j] m, \{v_1, ..., v_n\} = \mathcal{S}[S]\ l\ m \\
\mathcal{R}[\texttt{for} \langle V_1, V_2\rangle\ \texttt{in}\ S, R]\ h_0\ l\ m\ = h_n, \\
\quad h_j = \mathcal{R}[R]\ h_{j-1}\ l[V_1 \mapsto v_{1j}][V_2 \mapsto v_{2j}]\ m, \\
\quad \{\langle v_{11}, v_{21}\rangle, ..., \langle v_{1n}, v_{2n}\rangle\} = \mathcal{S}[S]\ l\ m \\
\mathcal{R}[\texttt{for} V = E_1\ ..\ E_2, R]\ h_0\ l\ m\ = h_n . h_{j+1} = \mathcal{R}[R]\ h_j \\
\quad l[V \mapsto j + (\mathcal{E}[E_1]\ h\ l\ m)]\ m,\ n = (\mathcal{E}[E_2]\ h\ l\ m) - (\mathcal{E}[E_1]\ h\ l\ m) \\
\mathcal{R}[G \Rightarrow C]\ h\ l\ m\ = \neg \mathcal{G}[G]\ h\ l\ m \mid \mathcal{C}[C]\ h\ l\ 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_1]\ h\ l\ m\ = !(\mathcal{G}[G_1]\ h\ l\ m) \\
\mathcal{G}[E_1 = E_2]\ h\ l\ m\ = (\mathcal{E}[E_1]\ h\ l\ m) == (\mathcal{E}[E_2]\ h\ l\ m) \\
\mathcal{G}[E_1 < E_2]\ h\ l\ m = (\mathcal{E}[E_1]\ h\ l\ m) < (\mathcal{E}[E_2]\ h\ l\ m) \\
\mathcal{G}[true]\ h\ l\ m = true \\
\mathcal{C}[HE.field = E]\ h\ l\ m = \langle \mathcal{HE}[HE]\ h\ l\ m, field, \mathcal{E}[E]\ h\ l\ m\rangle \in h \\
\mathcal{C}[HE.field[E_1] = E_2]\ h\ l\ m = \\
\quad \langle \mathcal{HE}[HE]\ h\ l\ m, field, \mathcal{E}[E_1]\ h\ l\ m, \mathcal{E}[E_2]\ h\ l\ m\rangle \in h \\
\mathcal{C}[V = E]\ h\ l\ m = (l(V) == \mathcal{E}[E]\ hl\ m) \\
\mathcal{HE}[V]\ h\ l\ m\ = l(V) \\
\mathcal{HE}[HE.field]\ h\ l\ m\ = b.\langle \mathcal{HE}[HE]\ h\ l\ m, field, b\rangle \in h \\
\mathcal{HE}[HE.field[E]]\ h\ l\ m\ = \\
\quad b.\langle \mathcal{HE}[HE]\ h\ l\ m, field, \mathcal{E}[E]\ h\ l\ m, b\rangle \in h \\
\mathcal{E}[V]\ h\ l\ m = l(V) \\
\mathcal{E}[number]\ h\ l\ m = number \\
\mathcal{E}[V.R_]\ h\ l\ m\ = b.\langle V, b\rangle \in m(R) \\
\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{E}[string]\ h\ l\ m = string \\
\mathcal{E}[\texttt{size}(SE)]\ h\ l\ m = \mid \mathcal{SE}[SE]\ l\ m \mid \\
\mathcal{E}[\texttt{element}\ E\ \texttt{of}\ SE]\ h\ l\ m = \text{given some ordering of } \mathcal{SE}[SE]\ l\ m, \\
\quad \text{pick element number } \mathcal{E}[E]\ h\ l\ m \\
\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 R\} \\
\mathcal{SE}[R.V]\ l\ m\ = \{y \mid \langle y, l(V)\rangle \in R\} \\
\mathcal{S}[S]\ l\ m = \{s \mid s \in m(S)\}
\end{array}$$

Figure 12: Denotational Semantics for External Constraints

In general, the repair action may need a source of new items to add to sets to bring them up to the specified size. Any supersets of the set (as specified using the model definition language from Section 3.1) are one potential source. For `struct`s, memory allocation primitives are another potential source. For primitive types, the action can simply synthesize new values. We allow the developer to specify which source to use and, in the absence of such guidance, use heuristics to choose a default source.

Note that the repair may fail if the system is unable to allocate a new `struct` (typically because it is out of memory) or find a new value within the specified range. Note also that the model definition language allows the developer to specify partition and subset inclusion constraints between the different sets in the model. When our implementation changes items in one set, it appropriately updates other sets to ensure that the model continues to satisfy these partition and subset inclusion constraints.

If S is a relation expression of the form $R.v$ or $v.R$, the repair action simply adds or removes tuples to satisfy the constraint. Note that because the items in the tuples must be part of the corresponding domain and range of the relation, a repair action that adds tuples to the relation may also need to add items to the domain or range sets of the relation. Repair actions that add tuples to relations therefore face the same issues associated with finding new items as the repair actions that add items to sets.

4.2.2 Inequality Propositions

Inequality propositions are of the form $V.R = E$, $!V.R = E$, $V.R < E$, $V.R <= E$, $V.R > E$, or $V.R >= E$. The repair actions calculate the value of E, then updates $V.R$ to be the closest value that satisfies the proposition.

4.2.3 Inclusion Propositions

Inclusion propositions are of the forms: V `in` SE where SE is a set in the model or a relation expression. The repair actions simply add or remove the value referenced by the label V to the set or the appropriate pair to the relation. This is done in a manner to satisfy the partition and subset requirements of the model definition.

4.2.4 Choosing The Conjunction to Repair

When faced with a choice of false conjunctions, the algorithm uses a cost function to choose which to repair. This cost function assigns a cost to each repair action; the cost of repairing a conjunction is simply the sum of the repair costs for all of its unsatisfied basic propositions. This approach is designed to minimize the number of changes made to repair the model. We have also tuned the repair costs to discourage the removal of objects from sets and tuples from relations. The idea is to preserve as much information from the original data structures as possible.

4.2.5 Termination

The repair action for one basic proposition may falsify another basic proposition. This raises the possibility that the repair algorithm may not terminate because of a cyclic repair chain. Conceptually, we eliminate this possibility by preanalyzing the specification to check that it can never generate any such cyclic chain.

The acyclicity checking algorithm first converts the body of each constraint into disjunctive normal form. It then constructs a *constraint dependence graph.* There is one node in the graph for each constraint and one node for each conjunction in the disjunctive normal form of each constraint. The graph contains the following edges:

- **Constraint to Conjunctions:** There is a directed edge from each constraint to each of its conjunctions.
- **Interference:** There is an edge from a conjunction to a constraint if applying an action to satisfy one of the basic propositions in the conjunction may falsify one of the basic propositions in one of the conjunctions of the constraint.

  The foundation of this construction is a procedure that determines if one basic proposition may *interfere* with another, i.e., if repairing the first proposition may falsify the second. The interference checking algorithm first checks if the two propositions involve disjoint parts of the model; if so, they do not interfere. If the two propositions may involve the same state, it reasons about the specific repair action and the second proposition. If the repair action is guaranteed to leave the model in a state that satisfies the second proposition, there is no interference. This is true if the first proposition implies the second. It may also be true even in some cases when the second proposition implies the first. For example, the two constraints `size`$(S) >= 1$ and `size`$(S) = 1$ do not interfere — the repair action for `size`$(S) >= 1$ makes `size`$(S) = 1$.

  Given this definition of interference, there is an edge from a conjunction to a constraint if one of the basic propositions from the conjunction interferes with one of the basic propositions from the constraint.
- **Quantifier Scope:** There is an edge from a conjunction to a constraint if repairing one of the basic propositions in the conjunction may add an object to a set or a tuple to a relation, and this addition may increase the scope of the quantifier in the constraint.

If the constraint dependence graph is acyclic, it is clear that the repair algorithm will terminate — once the first (in the topological sort order) violated constraint is repaired, it will never be falsified by the repair of any other constraint. Once the first has been repaired, the next constraint(s) (in the topological sort order), once repaired, will never be falsified, and so forth.

The termination checking algorithm first checks to see if the constraint dependence graph is acyclic. If it is not acyclic, it removes conjunctions from this graph in an attempt to make the graph acyclic. Note that it must leave at least one conjunction in the graph for each constraint. Once a conjunction is removed from the graph, it is marked as forbidden to ensure that the repair algorithm never chooses to repair an inconsistency by satisfying that conjunction.

In general, it may not be possible to produce an acyclic constraint dependence graph, in which case the termination checking algorithm rejects the specification. In practice, this does not seem to be a concern — the constraint dependence graphs for our benchmark applications are acyclic even without conjunction removal.

4.2.6 Relations in Expressions

It is possible for the specification to use a relation R in a context that requires the image of any item under the relation to be a singleton set. Examples of such contexts include arithmetic expressions of the form $E_1.R_1 + E_1.R_2$ and multiple relation dereferences of the form $E.R_1.R_2$. If the specification includes such *singleton* contexts, we require that the specification constrain the image of the relation to always have size 1.[2] Before evaluating any constraint that uses the relation in a singleton context, the repair algorithm first processes the constraints that force the image of all items in the domain of the relation to be a singleton.

4.2.7 Error Detection and Repair in External Phase

The algorithm detects violations of the external constraints by simply evaluating the constraints in the context of the model and the data structures. If a constraint is not satisfied, the algorithm has computed a set of quantifier variable bindings that falsify the constraint. i.e., that identify a value in the data structure that should be the same as a value computed using the model. In this case the repair algorithm simply assigns the data structure value to be the same as the model value.

The only potential complication is that different constraints may impose two different values on the same data structure value. We currently rely on the developer to provide specifications with at most one constraint for each data structure value. It is possible to develop algorithms that automatically check that specifications have this property.

4.3 Limitations

The goal of the repair algorithm is to deliver a model that satisfies the internal constraints and a combination of model and data structures that together satisfy the external constraints. We next summarize the situations in which the algorithm may fail to realize this goal.

The internal constraint repair algorithm will fail only because of resource limitations — i.e., if it is unable to find an item or tuple to add to a set or relation, either because it is unable to allocate a new `struct` or because there are no more distinct items in the set that it is using as a source of new items. The external constraint repair algorithm will fail only if the external constraints specify different values for the same data structure value — in this case, the algorithm will produce a data structure with only one of the values.[3]

The static cyclicity checks described in Sections 4.2.5 and 3.1 rule out many potential failure modes, in particular, they eliminate the possibility of unsatisfiable specifications.

5. EXPERIENCE

We next discuss our experience using our repair tool to detect and repair inconsistencies in data structures from several applications: a Linux file system, an interactive game, and Microsoft Office files.

[2] It is also possible to automatically augment the specification with these constraints.

[3] This discussion does not address failures caused by incorrect behavior on the part of the underlying computing infrastructure, for example corruption of the repair algorithm's data structures (this can be partially addressed by placing these data structures in a separate address space) or failure to notify the algorithm of changes in the accessibility of regions in the program's address space.

5.1 Methodology

For each application, we identified important consistency constraints and developed a specification that captured these constraints. We also developed a fault insertion strategy designed to simulate the effect of potential inconsistencies.[4] We applied the fault insertion strategy to the data structures in the applications, then compared the results of running a chosen workload with and without inconsistency detection and repair. We ran the applications on an IBM ThinkPad X23 with a 866 Mhz Pentium III processor and 384 MB of RAM. For the Linux file system and the interactive game application, we used RedHat Linux 7.2. For the Microsoft Office file application, we used Microsoft Office XP running on the Microsoft Windows XP operating system.

5.2 A Linux File System

Our Linux file system application implements a simplified version of the Linux ext2 file system [13]. The file system, like other Unix file systems, contains bitmaps that identify free and used disk blocks [6]. The file system uses these disk blocks to support fast disk block and inode allocation operations.

Our consistency constraints are that the inode bitmap block, the block bitmap block, the directory block, and the inode table block exist; that the inode bitmap is consistent with the use of inodes; that the block bitmap is consistent with the use of blocks; that blocks are not shared between files or other disk structures; that the file's size is consistent with the number of blocks in a file; that files contain only valid blocks; that inode reference counts are correct; and that directory entries refer to valid inodes. The specification contains 122 lines, of which 53 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.

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 remove and hardlink creation operations may leave inodes with incorrect reference counts; incomplete open operations that create new files may leave directory and inode entries in incorrectly initialized states.

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. For a file system with 1024 8KB blocks, our repair tool takes 1.5 seconds to construct the file system model, check the consistency of the model, and repair the file system.

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.

5.3 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 consistency constraints are that tiles have valid terrain values, a given city has exactly one location, cities are not in the ocean, and that the location of a city on the map is consistent with the location the city has recorded internally.

Our fault insertion strategy changes the terrain values in 20 randomly selected tiles in the map before the game starts. There are two possible errors: illegal terrain values or city 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 moving cities from tiles with illegal terrain types or oceans to tiles with a land type terrain. The specification consists of 218 lines, of which 173 lines contain structure definitions. The primary obstacle to 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.

We used the built-in test mode to play 25 games. In each game we set a different seed for the random number generator, resulting in different but repeatable games. In all of these games, our repair tool was able to repair the introduced inconsistencies and the game was able to execute without failing (although the game played out differently because of changed terrain values). Without repair, the game always crashed with a segmentation fault caused by indexing an array with an illegal terrain value. In this application, our repair tool takes 6.7 seconds to construct the model, check its consistency, and repairing the game map.

In addition to incorrect terrain values, the algorithm is able to repair inconsistencies in the location of cities in the game. If necessary, it removes extra city references to ensure that each city is referenced by only one tile and changes the internally recorded location of each city to ensure that it is consistent with the city's location on the map. The repair algorithm is limited in that if the entries describing several

[4] Fault insertion was originally developed in the context of software testing to help evaluate the coverage of testing processes [14]. It has also been used by other researchers for the purposes of evaluating standard failure recovery techniques such as duplication, checkpointing, and fast reboot [1]. The rationale behind fault insertion is that faults, while serious when they do occur, occur infrequently enough to seriously complicate the experimental investigation of failure recovery techniques. Fault insertion makes it practical to evaluate proposed recovery techniques on a range of faults.

basic aspects of the data layout (such as the size of the map) become corrupted, the system is not able to repair the map. Additionally, there are consistency conditions involving precalculated values, unit locations, and the map that are not well documented and not covered by the specification. As a result, there is some chance that the game may crash even after repair.

5.4 Microsoft Office File Format

Microsoft Office files consist of several virtual streams, each of which contains data for some part of the document. Each file also contains a FAT, which identifies the location of each stream within the file. Each virtual stream consists of a chain of blocks in the file. The file allocation table consist of an array of integers, with one integer per block in the file. For each block in the file, these integers indicate which block is next in the chain or whether the block is unused, terminates the chain, or stores part of the FAT.

Based on information available at http://snake.cs.tu-berlin.de:8081/ schwartz/pmh/, we developed a specification that captures the following consistency constraints: that blocks are not shared between chains, that the file has the correct number of FAT blocks for the its size, that FAT blocks are marked as such in the FAT, that the FAT contains valid block numbers, and that chains are appropriately terminated. The specification consists of 94 lines, of which 71 lines contain structure definitions. The availability of documentation made it straightforward to develop the specification.

Our fault insertion strategy injects one of four kinds of errors into the FAT: it can crosslink the ends of FAT chains (this causes blocks to be shared between streams), terminate FAT chains using an illegal block number, mark FAT blocks as unused, and mark the terminating block of a FAT chain as unused. The repair algorithm repairs crosslinked chains by terminating the chains immediately prior to the crosslinking. It repairs FAT chains that contain illegal block numbers by terminating the chain at the illegal block number. It also overwrites FAT values to ensure that FAT blocks are marked as used for the FAT, and removes unused FAT blocks from FAT chains.

Our workload consisted of several consistent Microsoft Word files. For each file, we used our fault insertion strategy to create four damaged files, one for each kind of error. We then attempted to load the files into Microsoft Word.

Word was able to successfully load all of the repaired files, although in some cases the combination of fault insertion followed by repair removed blocks from streams and changed the document. Word was also able to successfully load files in which FAT blocks are incorrectly marked as unused, but failed to load files with the three other kinds of damage. It instead responded with the error message “The document name or path is not valid.” For a 150KB file, our repair tool takes 8.4 seconds to construct the model, check the consistency of the model, and repair the file.

In addition to the repairs described above, the repair algorithm is able to allocate new FAT sectors as needed. Because our specification only covers FAT consistency constraints, there is no guarantee that the file satisfies any other consistency constraint. In particular, we suspect that the individual streams may have internal consistency constraints, although we did not observe any violation of these (hypothetical) constraints in our experiments.

5.5 Discussion

We found it relatively straightforward to develop the specifications for all of our applications once we had an understanding of the data structures. In particular, we developed all three of our specifications in the course of single week. During this week, we spent significant amounts of time understanding the Freeciv source code and debugging our implementation. We believe that the benefits of automatic inconsistency detection and repair are well worth the effort required to develop the specification.

In this paper, we have treated the inconsistency detection as just a necessary prerequisite for repair. But we believe that the inconsistency detector could be very useful on its own as a debugging aid. We know of many projects that manually develop data structure consistency detectors and use these detectors as a crucial part of the debugging infrastructure. Our specification-based approach should make it substantially easier to obtain these inconsistency detectors.

6. RELATED WORK

Software reliability has been an important area for many years. Most research has focused on preventing or eliminating software errors, with the approaches ranging from enhanced software testing and validation to full program verification. Software error detection has become an especially active area in recent years [3, 4, 9, 2].

In contrast, our research goal is to enable software to survive errors by restoring data structure consistency. The remainder of this section focuses on other error recovery techniques.

6.1 Manual Detection and Repair Systems

Researchers have manually developed several systems that find and repair data structure inconsistencies. File systems have many characteristics that motivate the development of such programs (they are persistent, store important data, and acquire disabling inconsistencies in practice). Developers have responded with utilities such as Unix fsck and the Norton Utilities that attempt to fix inconsistent file systems.

The Lucent 5ESS telephone switch and IBM MVS operating systems are two examples of critical systems that use inconsistency detection and repair to recover from software failures [10, 11]. The software in both of these systems contains a set of manually coded procedures that periodically inspect their data structures to find and repair inconsistencies. The reported results indicate an order of magnitude increase in the reliability of the system [7]. Researchers have also developed a domain-specific language for specifying these procedures for the 5ESS system [8]. The goal is to enhance the reliability and reduce the development time of the inconsistency detection and repair software.

These successful, widely used systems illustrate the utility of performing inconsistency detection and repair. We see our use of declarative specifications coupled with automatically generated detection and repair code as representing a significant advance over current practice, which relies on the manual development of the detection and repair code. Our approach enables the developer to focus on the important data structure constraints rather than on the operational details of developing algorithms that detect and correct violations of these constraints. We believe our specification-oriented approach will make it much easier to develop reliable inconsistency detection and repair software.

6.2 Self-Stabilizing Algorithms

Researchers in the area of self-stabilizing algorithms have developed specific distributed algorithms that eventually converge to a stable state in spite of perturbations [5]. Our research goal differs in that 1) we aim to provide a general-purpose, specification-based inconsistency detection and repair technology for arbitrary data structures (as opposed to designing individual algorithms with desirable constraints), and 2) we are willing to accept potentially degraded behavior as the price of obtaining this generality.

6.3 Traditional Error Recovery

Error recovery has been an important topic ever since the inception of computer science as a field. One standard approach avoids transient errors by simply rebooting the system; this is perhaps the most widely practiced form of error recovery. Checkpointing enables a system to roll back to a previous consistent state when it fails. Transactions support consistent atomic operations by discarding partial updates if the transaction fails before committing [7]. Database systems use a combination of logging and replay to avoid the state loss normally associated with rolling back to a previous checkpoint. In effect, the log serves as a redundant, very simple data structure that can be used to rebuild the more sophisticated internal database data structures whenever they become inconsistent. There has recently been renewed interest in applying many of these classical techniques in new computational environments such as Internet services [12].

Our approach differs from these classical approaches in that it is designed to repair inconsistent data structures in place and continue executing rather than roll back to a known consistent state. Our approach can enable systems to recover even from persistent errors such as file system corruption. Unlike approaches based on checkpointing and replay, it may preserve much of the volatile state and avoids the need for logging and replay. It can also keep a system going without the need to take it out of service while it is rebooting. Finally, our approach differs in that we do not attempt to recover to a state that a (hypothetical) correct program would produce. Instead, our goal is to recover to a state consistent enough to permit the continued operation of the program within its design envelope.

7. CONCLUSION

Data structure inconsistencies are an important source of software errors. Our implemented system attacks this problem by accepting a data structure consistency specification, then automatically detecting and repairing data structures that violate this specification. Our experience indicates that our system is able to deliver repaired data structures that enable the corresponding programs to continue to execute successfully within their designed operating envelope. Without repair, the programs usually fail.

As the field of computer science continues to mature, there is an increasing need to deliver systems that can continuously operate for very long, even unbounded, periods of time. Repair is a central aspect of almost all long-lived systems in other fields, and we believe that the development of effective repair technology is a necessary prerequisite for the construction of robust, long-lived computer systems. We therefore see our research as taking an important step toward the effective construction of robust, self-healing systems that can successfully recover from the damage that they will inevitably experience during their long lifetimes.

8. REFERENCES

[1] P. Broadwell, N. Sastry, and J. Traupman. FIG: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and self-MANaged Systems*, June 2002.

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

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

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

[5] E. W. Dijkstra. Self-stabilization in spite of distributed control. In *Communications of the ACM 17(11):643–644*, 1974.

[6] B. Goodheart and J. Cox. *The Magic Gargen Explained:The Internals of Unix System V Release 4: An Open Systems Design*. Prentice Hall, 1994.

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

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

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

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

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

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

[13] D. Poirier. Second extended file system. `http://www.nongnu.org/ext2-doc/` , Aug 2002.

[14] J. M. Voas and G. McGraw. *Software Fault Injection*. Wiley, 1998.