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 consistency constraints, then dynamically detects and repairs violations of these constraints, enabling the program to continue to execute productively even in the face of otherwise crippling errors. 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 inconsistent data structures and enable the program to continue to operate successfully.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software Verification;
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

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 consistency 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.

1.1 Intended Scope

This research places a high priority on continuing to execute the program even after concrete evidence that the execution has sustained at least one error. We recognize that this approach is clearly not appropriate for all computations — in many cases, the best response is to simply stop the execution and await external intervention, or reboot the system and restart from a clean state.

In some cases, however, continued execution is crucial. Our society is starting to deploy increasing numbers of safety-critical systems that interact directly with the physical world. Moreover, many of these systems rely completely on the active participation of the software for their continued safe execution, *even if the only goal of that execution is to bring the system down safely so that it can be repaired.* Embedded software often controls physical systems that operate for significant periods of time in active or unstable states. If the software fails, the resulting uncontrolled operation can cause catastrophic damage that seriously threatens human lives and property. In this context, even a partially impaired execution is far preferable to no execution at all.

Another broad class of systems monitors ongoing physical or information processes and presents summarized results to human users. The data structures in these systems typically reflect a sliding window of observations and predictions centered around the current time. Most data structure properties are transient, sliding through the system as it continually rebuilds its data structures to reflect its ongoing movement through time. In this context, any data structure anomalies will eventually be flushed out of the system as long as it continues to operate. Automatic data structure repair is one mechanism that can enhance the ability of these systems to execute through errors and eventually

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

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.

move back to a completely correct execution. Another rationale for continuing to execute is that the data structure corruption and repair may only affect a small part of the data and generated results in the system, with the other data and results still valid and useful as long as the program continues to execute. This is the case in our air-traffic control application.

Finally, many systems have persistent data structures (for example, file systems, application data files, or serialized data structures) whose inconsistencies persist across system restarts. By restoring key consistency properties, automatic data structure repair may enable tools or applications to work with or extract information from partially corrupted versions of these data structures.

1.2 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 involving 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.

1.3 Invoking Check and Repair

Our implemented system supports several mechanisms for invoking the consistency check and repair algorithm. One issue is that many correct data structure updates temporarily violate the consistency properties, then restore the properties as they complete. We must ensure that the check and repair does not interfere with such correct updates.

Our first mechanism is to simply enable the programmer to identify points in the program where he or she expects the data structures to be consistent. At each such point, the repair algorithm executes to find and repair any inconsistencies. An alternate mechanism augments the program to catch signals from faults such as divide by zero and segmentation fault violations. Because such faults are often caused by inconsistent data structures, the signal handler invokes the check and repair algorithm, then resumes the execution at the nearest consistency point. It is of course possible to use both of these mechanisms in the same program.

For persistent data structures, we generate a stand-alone version that reads in the data structure from persistent storage, repairs any consistency violations, then writes the data structure back out. This version can execute independently of other applications that access the data structure, or it can be integrated with these applications to perform the check and repair immediately after a data structure is written out or immediately before it is read back in.

1.4 Experience

We have used our tool to repair inconsistencies in four applications: an air-traffic control system (this system is in daily use in air-traffic control centers surrounding several major metropolitan airports), a simplified Linux file system, an interactive game, and Microsoft Word files. In this context, we have applied the tool to correct out of bounds array indices, 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 the correctness of recorded data structure sizes. Our tool is also able to correct corrupted 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 are very small in comparison with the size of the application and were relatively straightforward to develop once we understood the underlying data structures. We also found 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.

1.5 Other Applications of Our Specifications

We have presented our approach as valuable because of its ability to enable programs to recover and execute through errors, even though the execution may not correspond directly to the execution of a fully correct program. However, there are other advantages that developers may find useful.

First, developers may find the consistency checking functionality of our tool useful in its own right as a debugging aid to quickly find any inconsistencies generated by an incorrect program. For some deployed applications, the correct behavior is not to attempt to repair data structure inconsistencies and continue to execute, but to instead terminate or suspend as soon as an error is detected. Our consistency

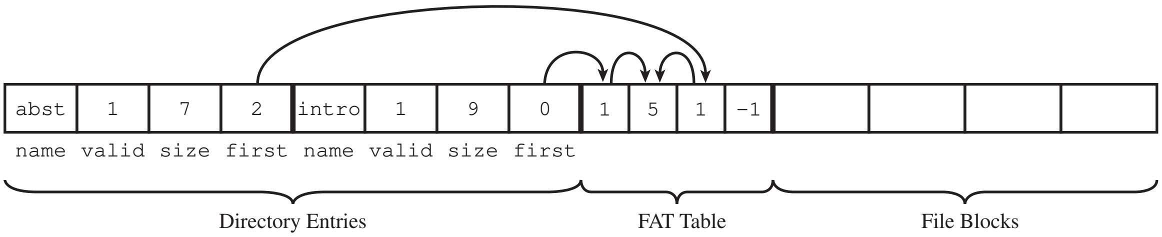

Figure 1: Inconsistent File System

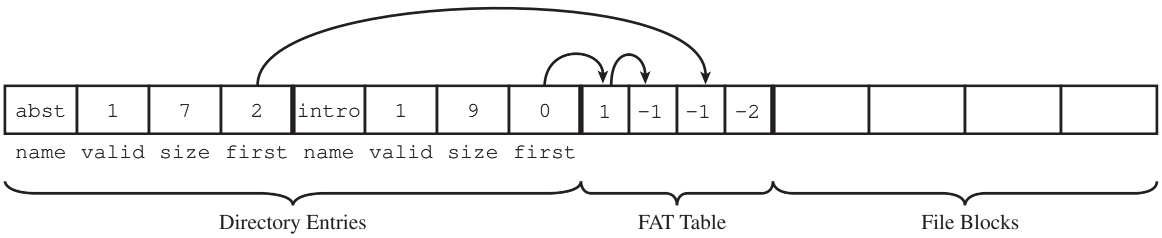

Figure 2: Repaired File System

checker can be used to implement this fail-stop behavior — as soon as it detects a data structure inconsistency, it can be configured to terminate or suspend the execution.

Second, our consistency checking and repair functionality can be used to provide insight into the behavior of the program. It can be configured to produce a trace of the inconsistencies it found and repair actions it took to eliminate the inconsistencies. The resulting logs may help developers identify errors and help narrow down the source of the errors in the program.

Finally, specifications of key data structure properties can also be useful as documentation or as a precise means of communicating the desired properties within or between teams developing the software.

1.6 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.

- **Specification Language:** It presents a new specification language that enables the developer to express key consistency properties of low-level, highly efficient data structures in a clean, general way.

- **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.

The remainder of the paper is structured as follows. Section 2 presents an example that we use to illustrate our approach. Section 3 presents the specification language used to express the consistency constraints. Section 4 presents the inconsistency detection and repair algorithms. Section 5 presents our experience using automatic data structure repair in several benchmark applications. Section 6 discusses related work; we conclude in Section 7.

2. EXAMPLE

We next present a simple file system example that illustrates how our technique works. The file system consists of three parts: the directory, the 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 active or not, a field indicating the size 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 size 7 and starts at file block 2; `intro` has size 9 and starts at file block 0.

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

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

Figure 3: Object and Relation Declarations

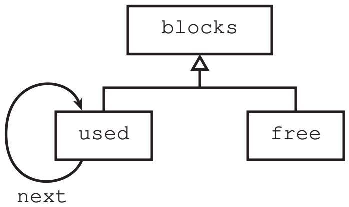

Figure 4: Graphical Representation of Object and Relation Declarations

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

Figure 5: Structure Declarations

```
Disk disk;

for i in 0..NumEntries, disk.table[i].valid &&
  disk.table[i].first < NumBlocks =>
  disk.table[i].first 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 6: Model Definition Declarations and Rules

of abstraction, we allow 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 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. Object modeling formalisms such as UML [26] and Alloy [18] have a graphical representation for such declarations; Figure 4 presents this representation for our example. The box labeled `blocks` represents the set of blocks, the box labeled `used` represents the set of used blocks, and the box labeled `free` represents the set of free blocks. The line with an empty arrowhead connecting the `used` and `free` boxes to the `blocks` box indicates that, together, `used` and `free` partition `blocks`. The `next` edge represents the `next` relation on `used` blocks.

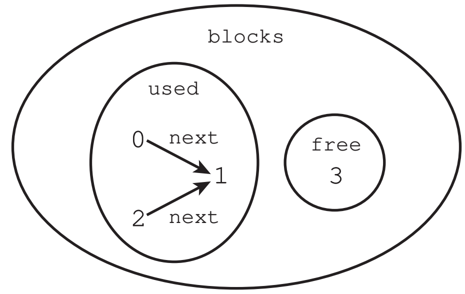

Figure 7: Inconsistent Model

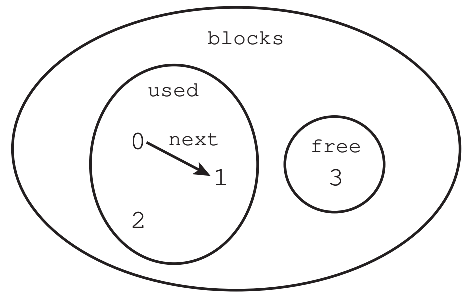

Figure 8: Repaired Model

Figure 5 presents the structure declarations for the file system. The `Entry` declaration identifies the format of each directory entry. The `name` field contains the name of the file, the `valid` field indicates whether the directory entry corresponds to a valid file or not, the `size` field gives the size of the file, and the `first` field is the index of the first disk block (and the index of the first FAT table entry) for the file. The `Disk` declaration identifies the disk as 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.

Figure 6 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 6 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 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.

When we apply the model construction rules in Figure 6 to our example file system in Figure 1, we obtain the model in Figure 7. This model has the following sets and relations: `used` $= \{$`0, 1, 2`$\}$, `free` $= \{3\}$, and `next` $= \{\langle 0, 1\rangle, \langle 2, 1\rangle\}$. The figure uses a Venn diagram to present the assignment of objects (in this case, `0,1,2,` and `3`) to sets; it uses arrows to represent the `next` relation.

2.2 Consistency Constraints

Internal constraints are stated using the model exclusively and not the concrete data structures. Figure 9 presents the single 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 9: Internal Consistency Constraint

In the model in Figure 7, file block 1 is in two chains — both $\langle 0, 1 \rangle$ and $\langle 2, 1 \rangle$ are in the `next` relation. This inconsistency violates the constraint that `size(next.1)<=1`. To repair this inconsistency, the repair algorithm will remove one of the tuples in the `next` relation. Figure 8 presents the repaired model — the repair algorithm has chosen to remove $\langle 2, 1 \rangle$ from the `next` relation.

External constraints may reference both the model and the concrete data structures. Figure 10 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 translate the model repairs back into 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 10: External Consistency Constraints

2.3 Repaired File System

Figure 2 presents the repaired file system from Figure 1. Note that because our example focuses on consistency constraints involving the linking structure implemented in the FAT table, all of the modifications are confined to this table. This repair has eliminated the sharing of file block 1 and truncated the `abst` 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.

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

3.1 Structure Definition Language

The structure definition language allows the developer to declare the layout of the data structures in memory. Figure 11 presents the grammar for this language. It allows the developer to declare fields of a structure that are 8, 16, and 32 bit integers; structures; pointers to structures; arrays of integers, packed booleans, structures, and pointers to structures. The array bounds can be either constants or expressions over program variables. The developer can declare that region of memory in a structure is reserved, indicating that it is unused. Finally, the structure definition language supports a form of structure inheritance. A substructure must have the same size and contain all of the same fields as the superstructure, but it may define new fields in areas that are unused in the superstructure.

$$
\begin{array}{rcl}
\textit{structdefn} & := & \mathtt{struct}\ \textit{structurename} \\
& & (\textit{subtypes structurename})\ \{\textit{fielddefn}^*\} \\
\textit{fielddefn} & := & \textit{type field};\ |\ \mathtt{reserved}\ \textit{type};\ | \\
& & \textit{type field}[E];\ | \\
& & \mathtt{reserved}\ \textit{type}[E]; \\
\textit{type} & := & \mathtt{boolean}\ |\ \mathtt{byte}\ |\ \mathtt{short}\ |\ \mathtt{int}\ |\ \textit{structurename}\ | \\
& & \textit{structurename}\ * \\
E & := & V\ |\ \textit{number}\ |\ \textit{string}\ |\ E.\textit{field}\ | \\
& & E.\textit{field}[E]\ |\ E - E\ |\ E + E\ |\ E/E\ |\ E * E
\end{array}
$$

Figure 11: Structure Definition Language

The structure definition language is similar to that of C. However, it supports wider range of primitive data types, provides a form of structure inheritance, and allows the developer to define inline, variable-length arrays. These extensions enable the developer to precisely specify the format of the elements in many heavily encoded data structures.

3.2 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 $\mathtt{S}_1, ..., \mathtt{S}_n$ which together partition `S`. Changing the `partition` keyword to `subsets` removes the requirement that the subsets $\mathtt{S}_1, ..., \mathtt{S}_n$ 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 $\mathtt{S}_1$ and $\mathtt{S}_2$.

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 12 presents the grammar for the model definition language.

Figure 13 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

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

Figure 12: Model Definition Language

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

$$
\begin{array}{l}
\mathcal{R}[\texttt{for } V \texttt{ in } S, C] \ h \ l \ m \ = \bigcup_{v \in m(S)} \mathcal{R}[C] \ h \ l[V \mapsto v] \ m \\
\mathcal{R}[\texttt{for } \langle V_1, V_2\rangle \texttt{ in } R, C] \ h \ l \ m \ = \bigcup_{\langle v_1, v_2\rangle \in m(R)} \\
\qquad \mathcal{R}[C] \ h \ l[V_1 \mapsto v_1][V_2 \mapsto v_2] \ m \\
\mathcal{R}[\texttt{for } V = E_1 \ .. \ E_2, C] \ h \ l \ m \ = \\
\qquad \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 \ = \neg(\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 = 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}[number] \ h \ l \ m \ = number \\
\mathcal{E}[E.field] \ h \ l \ m \ = b \text{ such that } \langle (\mathcal{E}[E] \ h \ l \ m), field, b\rangle \in h \\
\mathcal{E}[E_1.field[E_2]] \ h \ l \ m \ = \\
\qquad c \text{ such that } \langle (\mathcal{E}[E_1] \ h \ l \ m), field, (\mathcal{E}[E_2] \ h \ l \ m), c\rangle \in h \\
\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
\end{array}
$$

Figure 13: Denotational Semantics for Model Definition Language

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

Each model definition contains a set of model definition rules $C_1, ..., C_n$. 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.

3.3 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 a reference to the nested `struct`), or a pointer (in which case $E.\texttt{f}$ denotes a reference to the `struct` to which the pointer refers). So for example, one would express the standard doubly linked list constraint (that following the `next` pointer then the `prev` pointer leaves one back at the original list node) as `[forall` V_1 `in` S_1`]`, `[forall` V_2 `in` S_2`]`$, V_1.\texttt{next} = V_2 \Rightarrow V_2.\texttt{prev} = V_1$. 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 `struct`s reside completely in allocated memory. In addition, if two `struct`s overlap, one must be completely contained within the other and the declarations of both `structs` must agree on the format of the overlapping memory. This approach ensures that only valid `struct`s appear in the model.

A final complication is that expressions of the form $E.\mathtt{f}.\mathtt{g}$ may appear in guards. If $E.\mathtt{f}$ is not valid, $E.\mathtt{f}.\mathtt{g}$ 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, $\langle E_1, E_2\rangle$ in R) predicates involving undefined values have the special value `maybe`. We use three-valued logic to evaluate guards involving `maybe`.

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

3.4 Internal Constraints

Figure 14 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 15 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 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 as output. When 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 3.3: 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.

We intend developers to use the internal constraint language to express the key consistency constraints. This language is oriented toward expressing local consistency properties rooted at objects within specific sets. It can therefore be difficult to specify global constraints involving large collections of objects.

It is possible to express ownership properties,[2] but expressing these properties requires the construction of auxiliary relations during the model construction phase. The acyclicity check in our current algorithm (see Section 4.2.5) currently rules out these kinds of specifications.

[2] Ownership properties capture encapsulation relationships between groups of objects [8, 2]. Conceptually, an ownership property might capture the requirement that all paths in the heap that lead to a given set of objects must go through an object that is the conceptual owner of all of the objects in the set.

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

Figure 14: Internal Constraint Language

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

$$
\begin{array}{l}
\mathcal{EV}[\mathtt{for}\ V\ \mathtt{in}\ S, C]\ l\ m\ = \\
\quad \bigwedge_{v \in m(S)} \mathcal{EV}[C]\ l[V \mapsto v]\ m \\
\mathcal{EV}[\mathtt{for}\ V = E_1..E_2, C]\ l\ m\ = \\
\quad \bigwedge_{v=\mathcal{E}[E_1]lm}^{\mathcal{E}[E_2]lm} \mathcal{EV}[C]\ l[V \mapsto v]\ m \\
\mathcal{EV}[B]\ l\ m = \mathcal{C}[B]\ l\ m \\
\mathcal{C}[!B]\ l\ m = \neg\mathcal{C}[B]\ l\ m \\
\mathcal{C}[B_1\ \mathtt{and}\ B_2]\ l\ m = \mathcal{C}[B_1]\ l\ m \wedge \mathcal{C}[B_2]\ l\ m \\
\mathcal{C}[B_1\ \mathtt{or}\ B_2]\ l\ m\ = \mathcal{C}[B_1]\ l\ m \vee \mathcal{C}[B_2]\ l\ m \\
\mathcal{C}[V\ \mathtt{in}\ SE]\ l\ m\ = l(V) \in \mathcal{SE}[SE]\ l\ m \\
\mathcal{C}[VE = E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m == \mathcal{E}[E]\ l\ m) \\
\mathcal{C}[VE < E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m < \mathcal{E}[E]\ l\ m) \\
\mathcal{C}[VE <= E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m \leq \mathcal{E}[E]\ l\ m) \\
\mathcal{C}[VE > E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m > \mathcal{E}[E]\ l\ m) \\
\mathcal{C}[VE >= E]\ l\ m\ = (\mathcal{V}[VE]\ l\ m \geq \mathcal{E}[E]\ l\ m) \\
\mathcal{C}[\mathtt{size}(SE) = C]\ l\ m\ = \mathcal{E}[\mathtt{size}(SE)]\ l\ m\ == C \\
\mathcal{C}[\mathtt{size}(SE) >= C]\ l\ m\ = \mathcal{E}[\mathtt{size}(SE)]\ l\ m\ \geq C \\
\mathcal{C}[\mathtt{size}(SE) <= C]\ l\ m\ = \mathcal{E}[\mathtt{size}(SE)]\ l\ m\ \leq C \\
\mathcal{V}[V.R]\ l\ m\ = y \text{ such that } \langle l(V), y\rangle \in m(R) \\
\mathcal{E}[\mathtt{size}(SE)]\ l\ m\ =\mid \mathcal{SE}[SE]\ l\ m \mid \\
\mathcal{E}[V]\ l\ m\ = l(V) \\
\mathcal{E}[E.R]\ l\ m\ = y \text{ such that } \exists z, z \in \mathcal{E}[E]\ l\ m \wedge \langle z, y\rangle \in m(R) \\
\mathcal{E}[E_1 \oplus E_2]\ l\ m\ = primop(\oplus, \mathcal{E}[E_1]\ l\ m, \mathcal{E}[E_2]\ l\ m) \\
\mathcal{SE}[S]\ l\ m\ = \{s \mid s \in m(S)\} \\
\mathcal{SE}[V.R]\ l\ m\ = \{y \mid \langle l(V), y\rangle \in m(R)\} \\
\mathcal{SE}[R.V]\ l\ m\ = \{y \mid \langle y, l(V)\rangle \in m(R)\}
\end{array}
$$

Figure 15: Denotational Semantics for Internal Constraint Language

3.5 External Constraint Language

Figure 16 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 17 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.

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

Figure 16: External Constraint Language

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 $\texttt{size}(S) = C$, $!\texttt{size}(S) = C$, $\texttt{size}(S) >= C$, or $\texttt{size}(S) <= C$ where C is an integer constant and S can be one of the sets in the model or a relation expression of the form $R.v$ or $v.R$.

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.

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

$\mathcal{R}[\texttt{for}\ V\ \texttt{in}\ S, R]h\ l\ m\ = \bigwedge_{v \in m(S)} \mathcal{R}[R]hl[V \mapsto v]m$
$\mathcal{R}[\texttt{for}\ \langle V_1, V_2\rangle\ \texttt{in}\ R, R]\ h\ l\ m\ = \bigwedge_{\langle v_1, v_2\rangle \in m(R)}$
$\quad \mathcal{R}[R]\ h\ l[V_1 \mapsto v_1][V_2 \mapsto v_2]\ m$
$\mathcal{R}[\texttt{for}\ V = E_1\ ..\ E_2, R]\ h\ l\ m\ = \bigwedge_{v=\mathcal{E}[E_1]\ h\ l\ m}^{\mathcal{E}[E_2]\ h\ l\ m}$
$\quad \mathcal{R}[R]\ h\ l[V \mapsto v]\ m$
$\mathcal{R}[G \Rightarrow C]\ h\ l\ m\ = (\neg\mathcal{G}[G]\ h\ l\ m) \vee \mathcal{C}[C]\ h\ l\ m)$
$\mathcal{G}[G_1\ \texttt{and}\ G_2]\ h\ l\ m\ = (\mathcal{G}[G_1]\ h\ l\ m) \wedge (\mathcal{G}[G_2]\ h\ l\ m)$
$\mathcal{G}[G_1\ \texttt{or}\ G_2]\ h\ l\ m\ = (\mathcal{G}[G_1]\ h\ l\ m) \vee (\mathcal{G}[G_2]\ h\ l\ m)$
$\mathcal{G}[!G_1]\ h\ l\ m\ = \neg(\mathcal{G}[G_1]\ h\ l\ m)$
$\mathcal{G}[E_1 = E_2]\ h\ l\ m\ = (\mathcal{E}[E_1]\ h\ l\ m) == (\mathcal{E}[E_2]\ h\ l\ m)$
$\mathcal{G}[E_1 < E_2]\ h\ l\ m = (\mathcal{E}[E_1]\ h\ l\ m) < (\mathcal{E}[E_2]\ h\ l\ m)$
$\mathcal{G}[\texttt{true}]\ h\ l\ m = true$
$\mathcal{C}[HE.field = E]\ h\ l\ m = \langle \mathcal{HE}[HE]\ h\ l\ m, field, \mathcal{E}[E]\ h\ l\ m\rangle \in h$
$\mathcal{C}[HE.field[E_1] = E_2]\ h\ l\ m =$
$\quad \langle \mathcal{HE}[HE]\ h\ l\ m, field, \mathcal{E}[E_1]\ h\ l\ m, \mathcal{E}[E_2]\ h\ l\ m\rangle \in h$
$\mathcal{C}[V = E]\ h\ l\ m = (l(V) == \mathcal{E}[E]\ hl\ m)$
$\mathcal{HE}[V]\ h\ l\ m\ = l(V)$
$\mathcal{HE}[HE.field]\ h\ l\ m\ = b$ such that $\langle \mathcal{HE}[HE]\ h\ l\ m, field, b\rangle \in h$
$\mathcal{HE}[HE.field[E]]\ h\ l\ m\ =$
$\quad b$ such that $\langle \mathcal{HE}[HE]\ h\ l\ m, field, \mathcal{E}[E]\ h\ l\ m, b\rangle \in h$
$\mathcal{E}[V]\ h\ l\ m = l(V)$
$\mathcal{E}[number]\ h\ l\ m = number$
$\mathcal{E}[V.R]\ h\ l\ m\ = b$ such that $\langle V, b\rangle \in m(R)$
$\mathcal{E}[E_1 \oplus E_2]\ h\ l\ m = primop(\oplus, (\mathcal{E}[E_1]\ h\ l\ m), (\mathcal{E}[E_2]\ h\ l\ m))$
$\mathcal{E}[string]\ h\ l\ m = string$
$\mathcal{E}[\texttt{size}(SE)]\ h\ l\ m = \mid \mathcal{SE}[SE]\ l\ m \mid$
$\mathcal{E}[\texttt{element}\ E\ \texttt{of}\ SE]\ h\ l\ m =$given some ordering of $\mathcal{SE}[SE]\ l\ m$,
$\quad$pick element number $\mathcal{E}[E]\ h\ l\ m$
$\mathcal{SE}[S]\ l\ m = \{s \mid s \in m(S)\}$
$\mathcal{SE}[V.R]\ l\ m\ = \{y \mid \langle l(V), y\rangle \in R\}$
$\mathcal{SE}[R.V]\ l\ m\ = \{y \mid \langle y, l(V)\rangle \in R\}$

Figure 17: Denotational Semantics for External Constraint Language

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 3.2) 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 update $V.R$ to be the closest value that satisfies the proposition.

4.2.3 Inclusion Propositions

Inclusion propositions are of the form 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 to repair, 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 objects and/or relations, it reasons about the specific repair action and the second proposition. If the repair action is guaranteed to leave the model in a state that satisfies the second proposition, there is no interference. This is true if the first proposition implies the second. It may also be true even in some cases when the second proposition implies the first. For example, the two constraints $\texttt{size}(S) >= C$ and $\texttt{size}(S) = C$ do not interfere — the repair action for $\texttt{size}(S) >= C$ makes $\texttt{size}(S) = C$.

  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), 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 prunes 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.[3] 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. 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 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.

4.3.1 Repair Costs

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

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

We have isolated the choice of which violated constraint to repair inside a separate procedure in our implementation. It is straightforward to allow the developer to provide us with a partial implementation of this procedure — each time there is a choice to be made, our system would invoke the developer's implementation, which would return a subset of the choices that it found acceptable. Our system would then use the repair costs to choose the least costly alternative from within that subset. In principle, this mechanism gives the developer complete control over the choice should he or she choose to exert this control. It would even be possible to extend the system to enable the developer to specify that, in the current state of the computation, there is no need to repair a given violated constraint.

4.3.2 Set Membership Changes

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

Note that our specifications also allow partition constraints, which specify that a collection of subsets must partition another set. Membership changes in one of the sets often entail membership changes in some other sets. For example, when a repair action adds a new object to the partitioned set, it must also add that object to one of the subsets that partition the original set. In such cases, we allow the developer to control which sets objects are added to or removed from to satisfy the partition constraints.

4.3.3 Hand-Coded Repair Routines

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

4.3.4 Critical Constraints

In some cases, the developer may wish to identify *critical constraints*, or constraints that are so crucial to the continued successful execution of the program that if they are violated, the best strategy is to simply terminate or suspend the execution and await external intervention. We allow the developer to flag such constraints in the specification. If the consistency checker finds that a critical constraint is violated, it suspends the program. It is straightforward to generalize this technique to allow the developer to specify an arbitrary sequence of actions to be executed when a critical constraint is violated. This mechanism would allow the developer to specify a "safe exit strategy" designed to bring the system safely down to a quiescent state.

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

4.4 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.

4.4.1 Static Cyclicity Checks

The static cyclicity checks described in Sections 4.2.5 and 3.2 rule out many potential failure modes, in particular, they eliminate the possibility of unsatisfiable specifications. They also prevent the expression of several classes of constraints. As discussed in Section 3.4, they rule out constraints involving ownership properties. They also rule out collections of constraints whose repair actions involve both insertions and removals from the same set or relation. Consider, for example, a specification that requires 1) every node in a list to refer to a non-null element and 2) every such element to have at most one incoming reference from such a node. Assume also that the repair action for the first constraint chooses an arbitrary element and makes the empty list node refer to that element, while the repair action for the second constraint simply removes excess incoming references. With these repair actions, the constraint dependence graph contains a cycle and the static cyclicity checks would reject the specification.

One way to extend the approach to handle such constraints is to synthesize coarser granularity repair actions that do not cause cascading constraint violations. In the example above, it is possible to eliminate extra incoming references to list elements from list nodes by choosing a node where such a reference originates, allocating a new element, then redirecting the node to refer to the new element. This repair action removes excess incoming references without causing a node to fail to refer to some element. It therefore eliminates the interference between the two constraints, removing the cyclicity in the constraint dependence graph.

4.4.2 External Constraints

As currently formulated, it is the responsibility of the developer to ensure that the external consistency constraints correctly translate the model repairs back into the concrete data structures. If the developer does not define the external consistency constraints correctly, the repair algorithm may fail to leave the data structures in a consistent state. In particular, the reapplication of the model definition rules to the repaired state may fail to produce a consistent model.

It is possible to eliminate the external consistency constraints by applying goal-directed reasoning to the model definition rules to automatically translate the model repairs back into data structure repairs. This extension would simplify the specifications and eliminate the possibility of incorrect external constraints failing to correctly translate the model repairs back into the data structures.

4.4.3 Underlying Computing Infrastructure

To this point, we not addressed failures caused by incorrect behavior on the part of the underlying computing infrastructure, for example corruption of the repair algorithm's data structures. One way to address this issue is to place these data structures in a separate address space not accessible to the application. It may also be possible to attack this kind of corruption by (recursively) applying data structure repair to the repair algorithm's data structures.

4.5 Enhancements

We next discuss several improvements and enhancements to our current data structure repair algorithm.

4.5.1 Repair Guarantees

Although the current repair algorithm attempts to minimize the number of actions required to perform the repair (which should, in turn, leave the repaired state at least heuristically close to the starting inconsistent state), there is no guarantee (other than consistent data structures) that characterizes its behavior. Developers contemplating the use of data structure repair may find themselves uncomfortable with this situation and may wish to obtain a better understanding of the consequences of using data structure repair in their system. Additional guarantees may help them obtain this understanding and increase their confidence in using this technique.

Information preservation guarantees characterize the effect of the repair actions on the information stored in the data structures. For example, an analysis of the repair algorithm for a given specification may be able to provide a guarantee that the repair actions will never remove an object from a set or a pair from a relation. An extension might guarantee that any removal of a pair from a relation would be followed by an insertion of a replacement pair with one or both of the objects in the original pair replaced by copies. One could even use an algorithm similar to that described in Section 4.2.5 to prune repair actions that would violate these guarantees. These kinds of guarantees could (depending on the meaning of the data structures) assure the developer that the repairs will never destroy information present in the inconsistent data structures.

Propagation guarantees characterize how far cascading repair actions may propagate. For example, an analysis of the repair algorithm for a given specification may be able to guarantee that the repair actions for an inconsistency in one part of the data structure will never propagate to change another part of the data structure. The analysis may also be able to bound the number of actions required to repair a given inconsistency; this bound limits how far the effect of the repair may propagate. Such propagation guarantees may assure the developer that the application of data structure repair in one part of the system will not interfere with the operation of another part of the system. They may also help the developer understand the potential impact of using data structure repair to eliminate specific classes of inconsistencies that have been observed to occur in practice.

4.5.2 Performance Improvements

Our current implementation uses an interpreter to construct the model, check for consistency violations, and repair the violations. As mentioned in Section 5, the performance of this implementation is adequate for our set of benchmark applications. However, it is possible to deliver an implementation with substantially better performance. We are currently exploring three alternatives: model elision, incremental checking, and check on access.

Model elision analyzes the specification to determine when it is possible to perform the consistency checking directly on the data structures without explicitly constructing an intermediate model. This technique would eliminate both the computation time and the memory overhead associated with building the model. The elimination of memory overhead may be especially important for embedded devices with small memories.

Incremental checking is designed for applications that repeatedly execute consistency checks. The idea is to track writes to the data structures, then use this information to check only those parts of the data structures whose consistency properties could have changed since the last check. We anticipate that this optimization could substantially reduce the checking overhead in programs with repeated consistency checks.

Our current checking algorithm checks the complete data structure. It may also be possible to automatically distribute the checks so that the algorithm performs only the checks required to ensure the consistency of those parts of the data structure that the next section of code to execute accesses. The distributed checks then come to behave more like manually generated assertions in that each check is tailored for the particular context in which it executes. One can also view this approach as the dual of incremental checking — incremental checking checks only those parts of the data structure that the previous section of code changed, while check on access checks only those parts of the data structure that the next section of code will read.

Note that all of these optimizations focus on improving the performance of the consistency checker — our current expectation is that, for most applications, repair will be required infrequently enough to make the performance of the repair algorithm less important. We expect, however, that model elision could also be used to improve the repair performance should it become desirable to do so.

5. EXPERIENCE

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

5.1 Methodology

We developed a complete implementation of the data structure repair tool. The implementation and consists of roughly 13,000 lines of C++ code. The source code for the tool and sample specifications are available at
http://www.cag.lcs.mit.edu/∼bdemsky/repair.

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 air-traffic control system, 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 CTAS

The Center-TRACON Automation System (CTAS) is a set of air-traffic control tools developed at the NASA Ames research center [1, 27]. The system is designed to help air traffic controllers visualize and manage the complex air traffic flows at centers surrounding large metropolitan airports.[5] In addition to graphically displaying the location of the aircraft within the center, CTAS also uses sophisticated algorithms to predict aircraft trajectories and schedule aircraft landings. The goal is to automate much of the aircraft traffic management, reducing traffic delays and increasing safety. The current source code consists of over 1 million lines of C and C++ code. Versions of this source code are deployed at seven of the 21 centers in the continental United States (Dallas/Ft. Worth, Los Angeles, Denver, Miami, Minneapolis/St. Paul, Atlanta, and Oakland) and are in daily use at these centers.

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

The CTAS software maintains data structures that store aircraft data. Our experiments focus on the flight plan objects, which store the flight plans for the aircraft currently within the center. 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 to build the corresponding flight plan data structure is a challenging activity.

Our fault insertion methodology attempts to mimic errors in the flight plan processing routine 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

[4]Fault insertion was originally developed in the context of software testing to help evaluate the coverage of testing processes [30]. It has also been used by other researchers for the purposes of evaluating standard failure recovery techniques such as duplication, checkpointing, and fast reboot [3]. 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.

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

specification captures the constraint that the flight plan indices must be within the bounds of the airport data array. The specification itself consists of 100 lines, of which 83 lines contain structure definitions. The primary obstacle to developing this specification was reverse engineering the source (which consists of over 1 million lines of C and C++ code) 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. We identified consistency points within the application, then configured the system to catch addressing exceptions, perform the consistency checks and repair in the fault handler, then restart from the last consistency point. Each consistency check and repair takes several milliseconds, which is an acceptable repair time in that it imposes no performance degradation that is visible in the graphical user interface that displays the aircraft information.

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 processing. 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, and continued execution enables the system to provide this functionality to the controller even in the presence of flight plan processing errors. Second, only the origin or destination airport of the plane whose flight plan triggered the error is affected. All other aircraft (during the recorded feed, the system is processing flight plans for several hundred aircraft) are processed with no errors at all, enabling the system to deliver useful trajectory prediction and scheduling functionality for those aircraft. And finally, once the aircraft in question leaves the center, its data structures are deallocated from the system, which is then back to a completely correct state. One improvement that would further improve the utility of the repaired system is a way to visually identify aircraft with repaired flight plan information. We are currently exploring ways to leverage existing GUI functionality to make this happen.

The standard alternative to repair is to fail and reboot. This solution is problematic for this application because rebooting the system can take several minutes as the system acquires enough flight plans and radar data history to make reasonable trajectory predictions. And for the particular error we explored in our experiments, rebooting is futile. When the system reacquires and attempts to process the flight plan that caused the preceding failure, it will simply fail again.

5.3 A Linux File System

Our Linux file system application implements a simplified version of the Linux ext2 file system [25]. The file system, like other Unix file systems, contains bitmaps that identify free and used disk blocks [12]. 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 constraints state 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. The repair algorithm first traverses the blocks and inodes in the file system to construct a model of the file system. It then uses the model to compute correct values for the bitmap blocks and reference counts.

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.

5.4 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 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 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. The map was configured to contain 4,000 tiles. 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 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 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.

5.5 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 consists 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 were 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."

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.

5.6 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 the specifications for all of our applications except CTAS 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 also developed the CTAS specifications in less than a week, with much of the time devoted to reverse engineering the code (with the help of the NASA engineers in the CTAS group). The specifications are very small relative to the size of the application (although bear in mind that our current set of specifications do not necessarily capture all of the important consistency properties). In general, we expect 1) the specifications to be small in comparison with the application and 2) the overhead of the specification development to be very small in comparison with the effort required to develop the application. We believe that the benefits of automatic inconsistency detection and repair, in combination with the other benefits of developing precise data structure specifications, are, in most cases, well worth the effort required to develop the specification.

The CTAS system illustrates some of the reasons why continued execution can be the best choice for some applications. The absence of repair makes the entire computation vulnerable to errors, even if the error would have no effect on much of the data and functionality of the system. Repair enables the program to continue to execute and generate useful results from the correct parts of the data and the unaffected parts of the computation. The file system and Word file applications also have this characteristic — even a small consistency violation can make it impossible to access the rest of the (undamaged) data stored in the data structure. Repair makes it possible to access this data. And in some cases (as in the air-traffic control application and our file system workload) repair followed by continued execution eventually flushes any anomalies out of the system to restore the data structures to a completely correct state.

In this paper, we have treated 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.

5.7 Performance

Tables 1 through 4 present some performance results for our system. Table 1 presents, for each application, the number of model definition rule applications in the model construction phase and the total sizes of the sets and relations in the resulting models. Table 2 presents the execution times required to construct these models. Table 3 presents the the total number of internal constraint rule evaluations and the execution times required to perform the internal consistency checks for each application. Table 4 presents the number of evaluated external consistency rules and the execution time required to enforce the external consistency constraints. All execution times are measured in milliseconds. Note that for CTAS, the consistency check is done on a per-flight basis and the reported numbers are for check and repair applied to a single flight, not all of the flights in the system.

As these performance numbers show, the majority of the time is spent in the model construction phase, suggesting that the model elision optimization discussed in Section 4.5.2 could substantially improve the performance. We note that our current implementation uses an interpreter to perform the consistency check and repair. Our initial investigations show that, even without model elision, it should be possible to reduce the consistency checking overhead by at least two orders of magnitude.

The performance of our current consistency check and repair algorithm is more than adequate for all of our current implementations. We anticipate, however, that the overhead may become more problematic for systems with larger amounts of state. The optimizations discussed in Section 4.5.2 should substantially reduce the overhead and increase the range of systems in which our technique can be productively applied.

Application	Number of model definition rule applications	Total size of sets (objects)	Total size of relations (tuples)
CTAS	20	8	2
File system	11720	3128	1954
Freeciv	63072	7537	15990
Word	139740	64	17

Table 1: Number of model rule applications and size of model

Application	Time to construct model (ms)
CTAS	4.2
File system	1,188.9
Freeciv	5,609.1
Word	7,189.5

Table 2: Time to construct model

Application	Internal constraint evaluations	Time to check internal constraints (ms)
CTAS	4	.09
File system	2384	16.6
Freeciv	16004	175.3
Word	28	0.2

Table 3: Number of checks and time to check and repair internal constraints

Application	External constraint evaluations	Time to enforce external constraints (ms)
CTAS	4	0.2
File system	3164	59.5
Freeciv	12001	171.4
Word	39	1.2

Table 4: Number of checks and time to enforce external constraints

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 [9, 10, 16, 7]. 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 [17, 22]. 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 [13]. Researchers have also developed a domain-specific language for specifying these procedures for the 5ESS system [15]. The goal is to enhance the reliability and reduce the development time of the inconsistency detection and repair software. The 5ESS system has also served as the platform for PRL5, a declarative constraint specification language [21], and its compiler, which generates code to automatically check the consistency of a relational database used to store some it its information [14]. The compiler can also generate, for each operation, the weakest precondition required to ensure that the operation preserves the consistency constraints. Although the generated code does not perform any repairs, the consistency checking alone is valuable enough to justify its presence.

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. It also places the field on a firmer foundation, since it is based on a set of properties that the repair algorithm is designed to deliver rather than on a set of hand-coded repair routines whose effect may be more difficult to determine.

6.2 Integrity Maintenance in Databases

Database researchers have developed integrity management systems that enforce database consistency constraints. One goal is to enable the system to incorporate the effects of a transaction that leaves the database in an inconsistent state — instead of aborting the transaction, the integrity management system repairs the state from the end of the transaction to eliminate any inconsistencies. These systems typically operate at the level of the tuples and relations in the database, not the lower-level data structures that the database uses to implement this abstraction.

One approach is to provide a system that assists the developer in creating a set of production rules that maintain the integrity of a database [6]. Each production rule consists of a triggering component and a repair action to execute when the rule is triggered. The system automatically generates the triggering components of the production rules, using a triggering graph to check if repairs will terminate. The system relies on the developer to provide the actual repair actions; if the developer incorrectly specifies a repair action, the system may fail to maintain the integrity of the database.

This approach has been extended to enable the system to automatically generate both the triggering components and the repair actions [5]; the resulting system can automatically generate repairs that insert or remove tuples to or from a relation. The specification language can express similar properties as our internal constraint language, but the termination analysis is less precise. For some constraints the system may generate production rules that fail to terminate. For example, the system cannot automatically generate terminating repairs for a system of constraints that require a relation to be a function, then further constrain this function. Because of differences in the repair algorithms, our system is able to enforce these kinds of constraints.

Researchers have also developed a database repair system that enforces Horn clause constraints and schema constraints (which can constrain a relation to be a function) [29]. The system includes an interactive tool, which can help developers understand the consequences of repairing constraint violations. Our system supports a broader class of constraints — logical formulas instead of Horn clauses. It also supports constraints which relate the value of a field to an expression involving the size of a set or the size of an image of an object under a relation. Finally, it uses partition information to improve the precision of the termination analysis, enabling the verification of termination for a wider class of constraint systems.

It is also possible to apply constraint enforcement to structured documents [23]. This system accepts a set of consistency properties expressed in first-order logic, generates a set of repair actions for each constraint, and then interactively queries the user to select a specific repair action for violated constraints. Because the system performs no termination analysis, it is possible for infinite repair cycles to occur.

6.3 Assertions and Exception Handlers

Many programming languages support assertions as a way for developers to manually code consistency checks and exception handlers as a way to provide code to execute when these checks fail. One way of viewing our research is that it provides an automatically generated consistency check and exception handler that together find and eliminate any consistency violations. One advantage of our approach is its complete coverage — it always checks all of the constraints over all of the data structures, ensuring that it catches any inconsistencies before they propagate (manually developed assertions typically test only locally checkable properties on an easily accessible region of the data structure). The declarative nature of our specifications also reduces coding effort and makes it easier to determine that the code checks the correct set of constraints.

We also note that it can be extremely difficult to manually develop assertions and exception handlers that always operate safely in the presence of arbitrarily corrupted data structures. Automatic generation makes it easier to ensure that the code performs all of the checks required to operate without inadvertent failure.

6.4 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 [11]. 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. In some cases, however, our data structure repair algorithm may make the global program behave in a self-stabilizing way. In particular, if the effect of the repair is eventually flushed out of the system (as in the CTAS application), the data structures eventually converge back to a state that has no trace of the error or the repair.

6.5 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 state when it fails. Transactions support consistent atomic operations by discarding partial updates if the transaction fails before committing [13]. 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 [24]. One of the techniques that arises in this context, recursive restartability, composes large systems out of many smaller modules that are individually rebootable [4]. The goal is to build systems in which faults can be isolated at the module level by rebooting.

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 previous state. This approach avoids several problems associated with checkpointing. One potential problem is that the checkpointed state may contain latent inconsistencies that become visible only long after they are introduced. As long as these inconsistencies are present in the checkpointed state, the execution will remain vulnerable to errors triggered by the inconsistency. Another potential problem is that the current operation may trigger the same error even after replacing the current state with a previous checkpoint. Note that it is possible to apply our techniques to improve checkpoint-based approaches, either by checking for consistency before checkpointing the current state, or by repairing inconsistent checkpoints.

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. In many cases, the system will, over the course of time, flush the effects of errors out of its data structures and return to a completely correct state.

6.6 Specification Languages

The core of our specification language is the internal constraint language. The basic concepts in this language (objects and relations) are the same as in object modeling languages such as UML [26] and Alloy [18], and the constraint language itself has many of the same concepts and constructs as the constraint languages for these object modeling languages, which are specifically designed, in part, to be easy for developers to use. In addition to these ease of use considerations, the relative simplicity of the basic object modeling approach facilitates the automatic repair process. Because all structural properties are expressed in terms of cardinality constraints involving sets of objects and relations, it is possible to repair violations of these constraints by simply removing or inserting objects or pairs of objects into sets or relations.

Standard object modeling approaches have traditionally been used to help developers express and explore high-level design properties. Our approach, in contrast, also had to establish a precise connection between the low-level, heavily encoded data structures that appear in many programs and the high-level properties captured in the internal constraint language. Our model construction and external constraint languages provide a formal and quite flexible connection between these data structures and the model. These languages may therefore serve as an important component of future design conformance systems, which check that a program conforms to its high-level design [19].

Note also that factoring the consistency check and repair process into model construction followed by model check and repair isolates the treatment of the low-level details of the data structure within the model construction and external constraint enforcement phases. This isolation enables the application of our general-purpose consistency checking and repair algorithms to the full range of efficient, low-level, heavily encoded data structures.

6.7 Typestate Systems

In typestate systems, the type of each object can change over time to reflect changes in its properties [28, 20]. One key issue in typestate systems is understanding which object attributes should contribute the typestate and how changes in these attributes should affect the typestate. In this context, we can view each set in our model as corresponding to a conceptual typestate that objects may traverse during their lifetimes in the computation. The inclusion of an object in a set during the model construction identifies that set as one of the object's current typestates. This approach suggests that values in the fields of the object and its referencing and reachability relationships with other objects should contribute to its typestate. This level of expressibility is crucial both for design conformance (because many design properties involve changing object typestates) and for ensuring the utility of our consistency checking and repair tool (because appropriate consistency properties are different for objects in different typestates). Using a coarser abstraction such as the class of the object, which does not change over time to reflect conceptual state changes, would preclude the expression and enforcement of many important consistency properties. Our classification approach therefore suggests that it might be beneficial for future static typestate systems to capture information about referencing and reachability properties of the objects.

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

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[3] 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.

[4] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 110–115, Schloss Elmau, Germany, May 2001.

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

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

[7] 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.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1998.

[9] 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.

[10] 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.

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

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

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

[14] T. Griffin, H. Trickey, and C. Tuckey. Generating update constraints from prl5.0 specifications. In *Preliminary report presented at ATT Database Day*, September 1992.

[15] 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.

[16] 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.

[17] 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.

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

[19] D. Jackson and M. C. Rinard. Software analysis: A roadmap. In *Proceedings of 22nd International Conference On Software Engineering (ICSE'00) - Future of SE Track*, June 2000.

[20] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, 2002.

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

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

[23] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proceedings of the 25th International Conference on Software Engineering*, May 2003.

[24] 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.

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

[26] Rational Inc. The unified modeling language. `http://www.rational.com/uml`.

[27] B. D. Sanford, K. Harwood, S. Nowlin, H. Bergeron, H. Heinrichs, G. Wells, and M. Hart. Center/tracon automation system: Development and evaluation in the field. In *38th Annual Air Traffic Control Association Conference Proceedings*, October 1993.

[28] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, January 1986.

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

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