Automatic Data Structure Repair for Self-Healing Systems

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 have developed a system that accepts a specification of key data structure constraints, then dynamically detects and repairs violations of these constraints, enabling the program to recover from otherwise crippling errors to continue to execute productively. We present our experience using our system to repair violated constraints in a simplified version of the ext2 file system and in the CTAS air-traffic control program. Our experience indicates that the specifications are relatively straightforward to develop and that our technique enables the applications to effectively recover from data structure corruption errors.

1. INTRODUCTION

Any system that operates successfully for an extended period of time inevitably sustains and must recover from some form of damage. Development errors make software systems vulnerable to self-inflicted damage that may cause the system to crash, corrupt key data structures, or otherwise execute unacceptably. Data structure corruption can become especially problematic for persistent data structures since the corruption persists across system reboots and, unless repaired, can permanently impair the ability of the system to execute acceptably.

This paper proposes a new approach to recovering from data structure corruption. We have developed a tool that accepts a specification of key data structure consistency properties [7]. It uses this specification to automatically detect and repair violations of these consistency properties, enabling the system to recover from the inconsistency and continue to execute successfully within its designed operating envelope. This technique promises to dramatically increase the ability of software systems to automatically detect and recover from data structure corruption errors without the need for external operator intervention.

Our tool supports several different usage scenarios. It can be used in stand-alone mode to repair persistent data structures. It can also be used to repair the volatile data structures of a running program, with the repair applied either on program demand or to recover from an execution error such as an addressing violation.

*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.

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 the data structure consistency properties and the reasoning required to repair any inconsistencies.

Each specification contains model definition rules, a set of internal consistency constraints, and a set of external consistency constraints. The model definition rules identify the different kinds of objects and relations in the abstract view and define a translation from the concrete data structure to the abstract model. The internal consistency constraints capture the consistency properties of the data structure; these constraints are expressed at the level of abstract objects and relations in the model. The external consistency constraints capture the relationship between the model and the concrete data structure; our tool uses the external consistency constraints to translate any repairs from the model back into the concrete data structure. 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.

We have developed a complete implementation of the data structure repair tool. The implementation consists of approximately 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.

2. FILE SYSTEM CASE STUDY

We next discuss how our approach can be used to automatically detect and repair inconsistencies in a simplified version of the ext2 Linux file system. Like many Unix file systems, this file system has a superblock, an inode for every file, and uses bitmaps to facilitate the allocation and deallocation of inodes and disk blocks.

2.1 Data Layout Declarations

The declarations in Figure 1 specify the layout of data for selected portions of the file system. The structure definition language is similar to C with extensions to support packed bit arrays, a form of structural inheritance, and variable sized arrays.

The `Disk` struct declaration specifies that the disk consists of an array of `Block` objects, with the `superblock` stored in the first disk block and the `groupblock` stored in the second disk block. The superblock defines the parameters of the disk: the size and number of the disk blocks in the file system, the number of inodes, and the inode for the root directory. The groupblock contains references to the inode table and to the inode and block bitmaps. There is a bit in the block bitmap for each block in the file system and a bit in the inode bitmap for each inode in the file system. If the block or inode is currently used, this bit is set to `true`. Otherwise, it is set to `false`.

This file system has many consistency properties and can become corrupted in many ways. We focus on the following properties:

1. **Presence of File System Structures:** Basic file system structures should be present.
2. **Bitmap Consistency:** The inode and block bitmaps should be consistent with the use of the inode and blocks on the disk.
3. **Reference Count Consistency:** An inode's reference count should be consistent with the number of directory entries referencing it.
4. **Free Counts Correct:** The counts for free blocks and inodes should be consistent.
5. **Block Usage Consistency:** A given block should be used by at most one disk structure.

Notice that these constraints are stated at the level of abstract concepts such as blocks and inodes and not at the level of bits on the disk. We believe that this is the natural way that developers think about such constraints and that they would like to express their consistency properties at this level of abstraction. The abstract data structure view allows the developer to think about the data structures at this level.

2.2 Object Model

Figure 2 presents the object and relation declarations for the abstract representation used in our example. This abstraction contains three main categories of objects: `Blocks`, `Inodes`, and `DirectoryEntries`. The first declaration in Figure 2 specifies that the abstract model uses integers to identify the `Blocks` (this simplifies the correspondence between the abstract blocks in the model and the concrete disk blocks) and that the set of `Blocks` is partitioned into `UsedBlocks` and `FreeBlocks`. The set of `UsedBlocks` is further partitioned into different sets corresponding to the uses of blocks in the file system. These sets are the `SuperBlock` set, the `GroupBlock` set, the `FileDirectoryBlock` set, the `InodeTableBlock` set, the `InodeBitmapBlock` set, and the `BlockBitmapBlock` set. The `FileDirectoryBlocks` set is further partitioned into the `FileBlocks` set and the `DirectoryBlocks` set. The use of partitions ensures that a block cannot be used simultaneously in two different disk structures. Similarly, we have chosen to represent `Inodes` in the abstract representation with their integer index in the inode table and appropriately partitioned this set. Finally, the `DirectoryEntries` set contains the set of directory entries in the disk. This set represents the used directory entries in the file system.

The model uses relations to capture important properties of the objects in the model and to represent relationships between the objects. So, the `blockstatus` relation captures information in the block bitmap by mapping blocks to the set {`Free`, `Used`}. Similarly, the `inodestatus` relation maps `Inodes` to the set {`Free`, `Used`}, capturing the information in the inode bitmap. The relation `referencecount` maps `Inodes` to the corresponding reference count. The relation `filesize` maps `Inodes` to their corresponding size in bytes.

The `contents` and `inodeof` relations capture relationships between the objects in the model. Specifically, `contents` maps `UsedInodes` to the `FileDirectoryBlocks` that contain the contents of the file or directory, and `inodeof` maps `DirectoryEntries` to their corresponding `UsedInodes`.

```
Disk disk;

struct Disk {
  Block  b[disk.superblock.NumberofBlocks];
  label b[0]: Superblock superblock;
  label b[1]: Groupblock groupblock;
}

struct Block {
  reserved byte[disk.superblock.BlockSize];
}

struct Superblock subtype of Block {
  int FreeBlockCount;
  int FreeInodeCount;
  int NumberofBlocks;
  int NumberofInodes;
  int RootDirectoryInode;
  int BlockSize;
}

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

Figure 1: Data Layout Declarations

```
set Blocks of integer: partition UsedBlocks | FreeBlocks
set UsedBlocks of integer: partition SuperBlock |
  GroupBlock | FileDirectoryBlocks | InodeTableBlock |
  InodeBitmapBlock | BlockBitmapBlock
set FileDirectoryBlock of integer: DirectoryBlocks |
  FileBlocks
set Inodes of integer: partition UsedInodes | FreeInodes
set UsedInodes of integer: partition FileInodes |
  DirectoryInodes
set DirectoryInodes of integer: RootDirectoryInode
set DirectoryEntries of DirectoryEntry:
relation blockstatus: Blocks -> string
relation contents: UsedInodes -> FileDirectoryBlocks
relation inodestatus: Inodes -> string
relation referencecount: Inodes -> integer
relation filesize: Inodes -> integer
relation inodeof: DirectoryEntries -> UsedInodes
```

Figure 2: Object and Relation Declarations

```
[], true => 0 in SuperBlock
[], true => 1 in GroupBlock
[], disk.groupblock.InodeTableBlock<
  disk.superblock.NumberofBlocks =>
  disk.groupblock.InodeTableBlock in InodeTableBlock
[], disk.groupblock.InodeBitmapBlock<
  disk.superblock.NumberofBlocks =>
  disk.groupblock.InodeBitmapBlock in InodeBitmapBlock
[for i in UsedInode, for itb in InodeTableBlock,
  for j=0 to 11], cast(InodeTable,disk.b[itb]).itable[i].
  Blockptr[j]<disk.superblock.NumberofBlocks and
  !cast(InodeTable,disk.b[itb]).itable[i].Blockptr[j]=0
  => cast(InodeTable,disk.b[itb]).itable[i].Blockptr[j]
  in FileDirectoryBlocks
[for j=0 to disk.superblock.NumberofBlocks-1],
  !(j in UsedBlocks) => j in FreeBlocks
```

Figure 3: Model Definition Declarations

```
[for u in UsedInodes], u.inodestatus=Used
[for f in FreeInodes], f.inodestatus=Free
[for i in UsedInodes], i.referencecount=
  sizeof(inodeof.i)
[for i in UsedInodes], i.filesize<=
  sizeof(i.contents)*8192
[],sizeof(RootDirectoryInode)=1
[for u in UsedBlocks], u.blockstatus=Used
[for f in FreeBlocks], f.blockstatus=Free
[for b in FileDirectoryBlocks],sizeof(contents.b)=1
[],sizeof(BlockBitmapBlock)=1 and sizeof(SuperBlock)=1
[],sizeof(InodeTableBlock)=1 and sizeof(GroupBlock)=1
[],sizeof(InodeBitmapBlock)=1
```

Figure 4: Internal Consistency Constraints

2.3 Model Construction

Given the declarations of the objects and relations, we are now in a position to present the model definition, which translates the concrete data structure into the abstract model. Figure 3 presents part of these model definition declarations, which our tool uses to construct the abstract model. The intention is that the key high level consistency constraints will be enforced in the abstract model. Low level validity constraints (for example, that block references are in a valid range) are checked during the translation process so that low level errors are not a concern at the model level.

The first set of declarations specifies the sets of `Blocks` in the file system. For example, the first two declarations set up the `SuperBlock` set and the `GroupBlock` set. The omitted declarations define the various sets of `Inodes` in the file system, the `DirectoryEntries` set, and the various relations in the abstract model. Our tool interprets these definitions to derive an algorithm that constructs the objects and relations in the model, setting the stage for the definition and enforcement of the consistency properties.

2.4 Internal Constraints

Internal constraints capture the consistency properties that can be expressed using the model alone. We anticipate that these constraints will typically be used to capture the most important structural constraints. Figure 4 presents the set of internal constraints in our example.

The first two constraints in Figure 4 ensure that the `inodestatus` relation is consistent with the use of the `Inodes`. The third constraint ensures that the `referencecount` function returns the number of times an `Inode` is referenced by the `inodeof` relation. The fourth constraint ensures that the `filesize` function is consistent with the number of `Blocks` an `Inode` has. The fifth constraint ensures that the file system has a root directory. The next two constraints ensure that the `blockstatus` relation is consistent with the use of the `Blocks`. The next constraint ensures that a given block is in at most one file or directory. And the remaining constraints ensure that various disk structures exist. As this example illustrates, the internal constraints typically capture the important structural properties of the data structures and their relationships, and ensure that various parts of the data structures are consistent with each other.

SuperBlock=$\{0\}$
GroupBlock=$\{1\}$
FileDirectoryBlock=$\{6,8\}$
BlockBitmapBlock=$\{3\}$
InodeTableBlock=$\{4\}$
InodeBitmapBlock=$\{\}$
FileBlocks=$\{6,...\}$
DirectoryBlocks=$\{8\}$
FreeBlocks=$\{2,5,7,...\}$
FileInodes=$\{0,1\}$
RootDirectoryInode=$\{2\}$
FreeInodes=$\{...\}$
DirectoryEntries=$\{\mathtt{D}_0,\mathtt{D}_1...\}$
blockstatus=$\{\langle 0,\mathtt{Used}\rangle, \langle 1,\mathtt{Used}\rangle, \langle 2,\mathtt{Free}\rangle, \langle 3,\mathtt{Used}\rangle, \langle 4,\mathtt{Used}\rangle, \langle 5,\mathtt{Used}\rangle, \langle 6,\mathtt{Used}\rangle, \langle 7,\mathtt{Free}\rangle, \langle 8,\mathtt{Used}\rangle, ...\}$
contents=$\{\langle 0,6\rangle, \langle 1,6\rangle, \langle 2,8\rangle\}$
inodestatus=$\{\}$
referencecount=$\{\langle 0,1\rangle, \langle 1,1\rangle, \langle 2,0\rangle\}$
filesize=$\{\langle 0,100\rangle, \langle 1,200\rangle, \langle 2,8192\rangle\}$
inodeof=$\{\langle \mathtt{D}_0,0\rangle, \langle \mathtt{D}_1,1\rangle\}$

Figure 6: Abstract Representation for Corrupted File System

InodeBitmapBlock=$\{2\}$
FreeBlocks=$\{5,7,...\}$
blockstatus=$\{\langle 0,\mathtt{Used}\rangle, \langle 1,\mathtt{Used}\rangle, \langle 2,\mathtt{Used}\rangle, \langle 3,\mathtt{Used}\rangle, \langle 4,\mathtt{Used}\rangle, \langle 5,\mathtt{Free}\rangle, \langle 6,\mathtt{Used}\rangle, \langle 7,\mathtt{Free}\rangle, \langle 8,\mathtt{Used}\rangle, ...\}$
contents=$\{\langle 0,6\rangle, \langle 2,8\rangle\}$
inodestatus=$\{\langle 0,\mathtt{Used}\rangle, \langle 1,\mathtt{Used}\rangle, \langle 2,\mathtt{Used}\rangle, ...\}$
filesize=$\{\langle 0,100\rangle, \langle 1,0\rangle, \langle 2,8192\rangle\}$

Figure 7: Changes to the Abstract Representation of the Corrupted File System

2.5 Inconsistency Detection and Repair

Figure 5 presents a diagram of an inconsistent file system — the index of the block containing the inode bitmap is corrupted and two inodes reference the same block.

The first step in the inconsistency detection and repair process is to use the layout declarations, the model declarations, and the model definition rules to construct the abstract model. For the corrupt file system shown in Figure 5 the tool would generate the sets and relations in Figure 6.

The abstract representation shown in Figure 6 violates many of the constraints in Figure 4. For example, the empty `InodeBitmapBlock` set violates the last constraint in Figure 4. Furthermore, the fact that the relation `blockstatus` has the tuple $\langle 2,\mathtt{Used}\rangle$ and the set `FreeBlocks` contains the block 2 violates the seventh constraint shown in Figure 4 — the block is not used for anything but is marked `Used`. The fact that the `contents` relation contains two tuples that reference block 6 violates the eighth constraint.

The inconsistency detection algorithm evaluates the internal constraints over the model. In our example, this evaluation uncovers the consistency violations described above.

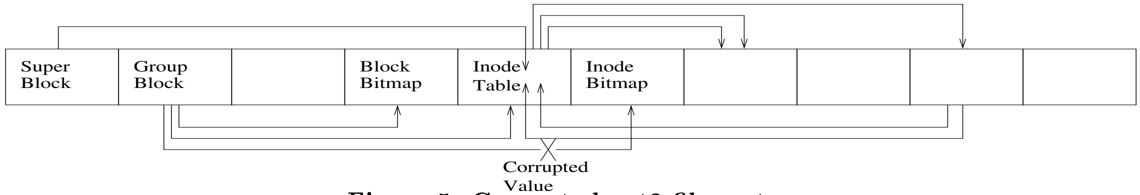

Figure 5: Corrupted ext2 file system

For each such violation, it identifies the violated constraint and the specific objects and relations that violate the constraint. For example, the tool discovers that (among other violations), block 6 is referenced by multiple inodes, indicating a violation of the constraint that each block is in at most one file or directory. When the tool discovers a violation, it executes a sequence of actions to repair the violation. It first converts the constraint into disjunctive normal form, i.e., a disjunction (ors) of conjunctions (ands) of basic propositions and negated basic propositions. The basic propositions capture basic numerical requirements on the values involved in relations (for example, a certain value must be less than a certain expression), constraints on the sizes of sets and relations, and objects or pairs that must be included in specific sets or relations. Each basic proposition comes with an action that is guaranteed to make the proposition true and an action that is guaranteed to make the proposition false. Depending on the form of the proposition, these repair actions may calculate a value that ensures that the constraint does or does not hold, or insert or remove objects or pairs from sets or relations.

To repair a violated constraint, the tool chooses one of the conjunctions in the normal form, then repairs all of the violated basic propositions in the conjunction. At this point the constraint is no longer violated, and the tool proceeds on to the next violated constraint.[1]

In our example, the tool discovers that the set `InodeBitmapBlock` is empty, which violates the last constraint in Figure 4. In this case, there is only one conjunction in the disjunctive normal form of the constraint; the tool must therefore insert an object into this set to satisfy the constraint. The repair action moves a block from the `FreeBlocks` set to the `InodeBitmapBlock` set (because the object and relation declarations in Figure 2 specify that `FreeBlocks` and `UsedBlocks` partition the set of blocks, `InodeBitmapBlock` is a `UsedBlock`, the repair action must remove the new `InodeBitmapBlock` from the set of `UsedBlocks`). The tool then moves on to repair the other violations, producing the repaired model in Figure 7.

2.6 External Constraints

External constraints constrain the relation between the abstract model and the concrete data structures. Figure 8 presents several of the external constraints for our example. The first four constraints ensure that any newly created structures in the model are written back out to disk. The next two constraints translate the model repairs of the `InodeTableBlock` back to the disk.

[1]The reader may be concerned that the repair process may not terminate. We have implemented a specification analysis algorithm that determines if the repair process will always terminate for a given specification; the tool uses this algorithm to reject any specifications that might generate repair sequences that do not terminate.

For our example, these constraints translate the repairs made in the abstract representation (shown in Figure 7) to the concrete data structures on the disk. The repaired version of the concrete data structure shown in Figure 5 is shown in Figure 9. Notice that our tool has regenerated the inode bitmap. Furthermore, the illegal block sharing has been removed, and the block bitmap is consistent with the use of blocks in the file system.

```
[for u in InodeTableBlock], true =>
  disk.groupblock.InodeTableBlock=u
[for u in InodeBitmapBlock], true =>
  disk.groupblock.InodeBitmapBlock=u
[for u in RootDirectoryInode], true =>
  disk.superblock.RootDirectoryInode=u
[for i in UsedInode, for itb in InodeTableBlock,
  for j=0 to 11], j<sizeof(i.contents) => cast(
  InodeTable, disk.b[itb]).itable[i].Blockptr[j]=
  element j of i.contents
[for i in UsedInode, for itb in InodeTableBlock,
  for j=0 to 11], !j<sizeof(i.contents) =>
  cast(InodeTable,disk.b[itb]).itable[i].Blockptr[j]=0
```

Figure 8: External Consistency Constraints

2.7 Experience

We developed a fault insertion strategy designed to simulate the effect of potential inconsistencies.[2] Our fault insertion mechanism 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. We crash the system part of the way through writing the files, then rerun the workload. The second run overwrites the partially written files and checks that the final versions are correct.

In all of our tested cases, the algorithm is able to repair the file system and the workload correctly runs to completion. Without repair, files end up sharing inodes and disk blocks and the file contents are incorrect. For a file system with 1024 8KB blocks, our repair tool takes 1.5 seconds on an IBM ThinkPad X23 with a 866 Mhz Pentium III processor and 384 MB of RAM running RedHat Linux 7.2 to construct the file system model, check the consistency of the model, and repair the file system.

[2]Fault insertion was originally developed in the context of software testing to help evaluate the coverage of testing processes [17]. It has also been used by other researchers for the purposes of evaluating standard failure recovery techniques such as duplication, checkpointing, and fast reboot [2]. 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.

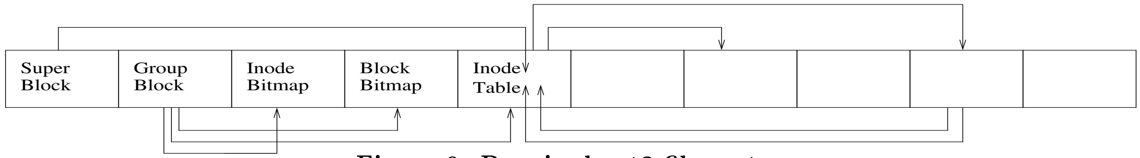

Figure 9: Repaired ext2 file system

3. CTAS CASE STUDY

The Center-TRACON Automation System (CTAS) is a set of air-traffic control tools developed at the NASA Ames research center [1, 16]. The system is designed to help air traffic controllers visualize and manage the complex air traffic flows at centers surrounding large metropolitan airports. 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 centers surrounding major metropolitan airports.

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 due to an addressing error. Our specification captures the constraint that the flight plan indices must be within the bounds of the airport data array. The specification itself consists of 100 lines, of which 83 lines contain structure definitions. The primary difficulty in developing this specification was understanding the flight plan data structures.

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 approximately 3 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.

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.

CTAS illustrates that data structure repair can enable systems to recover from otherwise fatal data structure corruption errors and enable the program to continue to execute successfully. This property may be especially important for safety-critical applications in which potentially degraded execution is far preferable to no execution at all.

The CTAS system in particular 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 the data and functionality of much 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. Note also that repair followed by continued execution may eventually flush any anomalies out of the system to restore the data structures to a completely correct state.

4. DEVELOPER CONTROL OF REPAIRS

The repair algorithm often has multiple options 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. Specifically, the developer may specify the costs of given repair actions, provide a procedure which decides which repair action to perform for a given constraint violation, or supply a hand-coded repair routine for a given constraint.

5. 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 [6, 10, 5]. In contrast, our research goal is to enable software to survive errors by repairing damaged data structures.

5.1 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 [11, 13]. 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 [8]. Researchers have also developed a domain-specific language for specifying these procedures for the 5ESS system [9]. The goal is to enhance the reliability and reduce the development time of the inconsistency detection and repair software.

5.2 Recovery Oriented Computing

Researchers in the area of recovery oriented computing have developed a variety of techniques to help software recover from runtime errors [14]. One of these techniques, 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.

In some cases, the consequences of an error may not be immediately apparent and the system may run ahead, generating an unacceptable execution. In such cases, the ability to undo an application's operations to return to an earlier state, repair the error in the earlier state, and then replay the application's operations would be useful. *Operation Undo* provides an application-neutral framework for building systems that support undo [3].

5.3 Specification Languages

The basic concepts in our internal constraint language are similar to those in constraint languages for object modeling formalisms such as UML [15] and Alloy [12]. Object models have traditionally been used to help developers explore conceptual design properties in the absence of any specific implementation. Our approach, in contrast, establishes a precise connection between the low-level, highly encoded data structures that appear in many programs and the high-level conceptual properties captured in our internal constraint language. This kind of connection may become especially important for future design conformance systems, which check that a program conforms to its design.

6. 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.

7. REFERENCES

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

[2] 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.

[3] A. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.

[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] 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.

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

[7] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. Technical Report MIT-LCS-TR-875, MIT, Massachusetts Institute of Technology, Dec. 2002.

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

[16] 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.

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