# Goal-Directed Reasoning for Specification-Based Data Structure Repair

Brian Demsky and Martin C. Rinard

**Abstract**—Software errors and hardware failures can cause data structures in running programs to violate key data structure consistency properties. As a result of this violation, the program may produce unacceptable results or even fail. We present a new data structure repair system. This system accepts a specification of data structure consistency properties stated in terms of an abstract set- and relation-based model of the data structures in the running program. It then automatically generates a repair algorithm that, during the execution of the program, detects and repairs any violations of these constraints. The goal is to enable the program to continue to execute acceptably in the face of otherwise crippling data structure corruption errors. We have applied our system to repair inconsistent data structures in five applications: CTAS (an air traffic control system), AbiWord (an open source word processing program), Freeciv (an interactive multiplayer game), a parallel x86 emulator, and a simplified Linux file system. Our results indicate that the generated repair algorithms can effectively repair inconsistent data structures in these applications to enable the applications to continue to operate successfully in cases where the original application would have failed. Without repair, all of the applications fail.

**Index Terms**—Testing and debugging, language constructs and features.

## 1 Introduction

Maintaining data structure consistency is a fundamental prerequisite for the acceptable execution of most software systems. Data structures that violate key consistency constraints can lead the software down unexpected execution paths, potentially causing the system to behave unacceptably or even fail. Unfortunately, there are many sources of inconsistent data structures—single event upsets that flip the values of bits in memory [37], unexpected interference from outside a given component of the software system, and overt errors such as data races, early exits from complex data structure updates, algorithmic oversights, and simple coding mistakes. The problem can be especially severe for data structures that persist across program executions—a single inconsistency in this kind of data structure can deny access to all of the information stored in the data structure.

The standard approach to dealing with data structure inconsistency is to work hard to prevent any inconsistencies from occurring in the first place. Approaches such as extensive testing, static analysis [21], [53], software model checking [14], error correction codes [45], and software isolation mechanisms [6] are all designed, in part, to eliminate as many potential data structure corruption errors as possible.

This paper presents a different and complementary approach for dealing with the data structure inconsistency problem. Instead of trying to eliminate all potential sources of corruption, our approach accepts the inevitability of some sources of data structure corruption. It therefore focuses on repairing corrupted data structures to restore acceptable system behavior. A developer using our approach first provides a specification of the data structure consistency constraints that the software relies on for its acceptable execution. Our compiler then processes this specification to automatically generate the *detection* and *repair* algorithms. The detection algorithm traverses the data structures to locate any violations of the consistency specification. The repair algorithm accepts as input an arbitrary data structure that violates the specification. It produces as output a repaired data structure that 1) is guaranteed to satisfy its specification and 2) is heuristically close to the original inconsistent data structure.

It is important to note that the goal of data structure repair is not to produce the same data structure that a (hypothetical) correct program would have produced. The goal is instead to produce a repaired data structure that enables the program to continue to execute acceptably. In our experience using data structure repair on a range of data structure corruption errors in our benchmark set of software systems, the repaired data structure always enabled the system to continue to execute acceptably. In the absence of data structure repair, the data structure corruption errors almost always caused the systems to fail.

Despite these results, there is no guarantee that the program will, in fact, continue to execute acceptably after the repair—the repair algorithm is guaranteed only to produce a data structure that satisfies its consistency specification, not a data structure that will always cause the program to continue to execute successfully. Any decision to use data structure repair should therefore take into account a comparative analysis of the consequences of simply halting the program as opposed to using data structure repair to enable continued execution. In general, these consequences will depend on the context into which

- *B. Demsky is with the Department of Electrical Engineering and Computer Science, University of California, Irvine, Irvine, CA 92697. E-mail: bdemsky@uci.edu.*
- *M.C. Rinard is with the MIT Computer Science and Artificial Intelligence Laboratory, The Stata Center, Building 32-G744, 32 Vassar Street, Cambridge, MA 02139. E-mail: rinard@lcs.mit.edu.*

*Manuscript received 16 Mar. 2006; revised 7 June 2006; accepted 28 Sept. 2006; published online 14 Nov. 2006.*
*Recommended for acceptance by W. Griswold and B. Nuseibeh.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0068-0306.*

the program is deployed. Halting eliminates the possibility of any further error propagation or damage caused by unacceptable program actions but also denies access to the functionality of the program and may lose the data in the damaged data structure. Data structure repair, on the other hand, may salvage much of the data in the damaged data structure and leave the functionality of the program available. Continued execution may be especially valuable for programs that control unstable physical phenomena —the Ariane 5 crash, for example, was caused by the guidance computer halting due to a failed integer conversion in the computation of an unused value [34].

### 1.1 Model-Based Approach

Our technique adopts a *model-based* approach with two views: a concrete view of the data structures as they are represented in the memory and an abstract view that (like object modeling formalisms such as UML [42] and Alloy [31]) models the data structures as sets of objects and relations between objects. A set of model definition rules translates the concrete data structures to the sets and relations in the abstract model. The key consistency constraints are expressed using the sets and relations in this model. The model definition rules encapsulate the data structure representation complexity and the consistency constraints encapsulate the complexity inherent in the consistency property.

Our approach provides several key benefits: 1) it provides a mechanism for separating objects that play different conceptual roles in a computation into different sets, allowing the developer to easily specify different constraints that apply to each of these different sets, 2) the model definition rules provide a clean, simple mechanism to specify data structure traversals, and 3) it provides a means to manage the complexity of data structure consistency properties.

### 1.2 Repair Algorithm

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

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

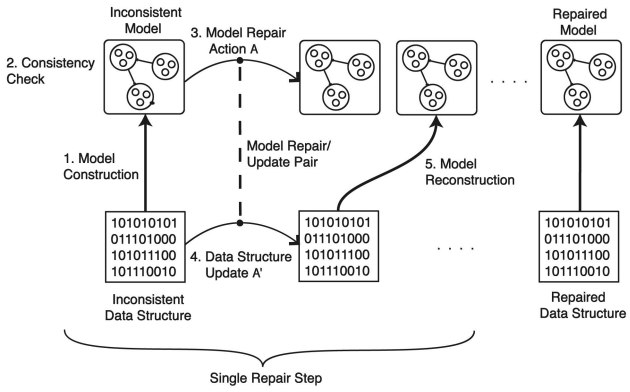


Fig. 1. Overview of repair process.

Our compiler uses a *repair dependence graph* to reason about the termination of the generated repair algorithm. The nodes in this graph represent constraints, repair actions, and changes to the sets and relations in the abstract model. The edges capture dependences between the constraints, repair actions, and the abstract model. The absence of certain kinds of cycles in the graph ensures that all repairs will terminate. In addition to analyzing the graph to determine termination, our compiler uses reasoning and search to remove (subject to certain structural requirements that the graph must satisfy) certain nodes, which correspond to possible ways that a constraint could be satisfied, to eliminate undesirable cycles. These removals further constrain the actions of the generated repair algorithm and ensure that the repair algorithm will never choose a repair strategy that leads to an infinite repair loop. The absence of infinite repair loops implies that the repair algorithm will successfully repair any violation of the consistency constraints.

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

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

updates that force all of these properties to hold. The effect is to remove the object from the set.

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

### 1.3 Expectations and Scope

The dependable computing community has developed a taxonomy of basic concepts in dependable and secure systems [9], and it is possible to analyze our approach in the context of this taxonomy. Data structure repair becomes relevant when a *fault* (such as an incorrect piece of code) is *activated* (by executing the incorrect code), leaving the data structures in an inconsistent state. This inconsistent state is an *error* which, unless corrected, may cause the program to observably deviate from its correct behavior. This deviation from correct behavior is called a *failure*.

In general, the inconsistent state may be any state that violates the system's specification. We distinguish two kinds of inconsistent states: 1) those that violate the data structure invariants and 2) those that satisfy the data structure invariants but are inconsistent with the input that the system has processed. Data structure repair, as implemented in our current system, is relevant only for inconsistent states that violate the data structure invariants. It is possible to increase the scope of our approach to record information about the input and include that information in the detection and repair of inconsistent data structures.

The goal of data structure repair is to ameliorate the effect of the errors by updating the data structures to eliminate any inconsistencies that the corresponding faults introduced. There are several possibilities:

- **Correction.** In some cases, the repair may leave the data structures in the same correct state as a (hypothetical) correct program would have left them when presented with the same input. In these cases, the repair makes it possible for the program to continue correctly (until it encounters another fault). This kind of correction is, in general, possible only when the data structure contains enough redundant information to successfully reconstruct any missing or corrupted parts of the data structure.
- **Restoration.** In other cases, the repair may leave the data structures in the same state as a (hypothetical) correct program would have left them, but only when presented with a different input. This may occur if the fault destroyed data, leaving the repair algorithm without enough information to correct the error. In these cases, the repair makes it possible for the program to continue to execute, with the continued execution producing outputs that would be correct for the different inputs but potentially not correct for the actual input.
- **Patching.** In yet other cases, the repair may leave the data structures in a state that satisfies the preconditions of the various software components that will access the data structures, but that no execution of the (hypothetical) correct program would ever produce. In these cases, the system will continue to execute but may produce outputs that no correct execution would ever produce. Note that this phenomenon can occur only when the consistency specification is incomplete in the sense that it is missing some constraints that all correct states satisfy. These constraints may be missing because the developer did not state the constraints explicitly or because the specification language was not expressive enough to state the constraint.
- **Updating.** If the consistency specification lacks certain crucial constraints, the repair may leave the data structures in a state that fails to satisfy some of the preconditions that must hold for components that access the data structures to execute successfully. In these cases, the system may encounter a fatal error. The constraints may be missing either because the developer failed to state the constraint or because the specification language was not expressive enough to state the constraint.
- **Overcorrection.** It is also possible for the consistency specification to contain constraints that some correct states violate. In these cases, the repair algorithm will update the data structures so that they satisfy these constraints, potentially interfering with the correct execution of the program.

As this discussion indicates, it is possible to view data structure repair as a specific instance of software fault tolerance in which the error detection phase consists of examining the data structures for inconsistencies and the recovery phase consists of updating the data structures to eliminate the inconsistencies. It differs from many software fault tolerance approaches in that the goal of the recovery phase is not necessarily to completely eliminate the error. Instead, one of the primary goals is to eliminate fatal errors that would otherwise cause the system to simply terminate and fail to deliver any service whatsoever.

There are obviously some situations in which users would prefer fail-stop behavior in which the system simply stops and awaits external intervention at the first sign of a fault. The choice of whether to stop or to repair and continue depends, in large part, on the context in which the system is used. Aspects that may play an important role include the feasibility and cost of providing external intervention, the acceptability of partially or even completely incorrect outputs, and the severity of the consequences of terminating the execution of the system. Consider, for example, a real-time or safety-critical system that controls unstable physical phenomena and whose termination therefore results in a disaster. If timely external intervention to recover from fatal errors is not feasible, the use of data structure repair (and other techniques designed to keep systems executing through otherwise fatal errors) may well

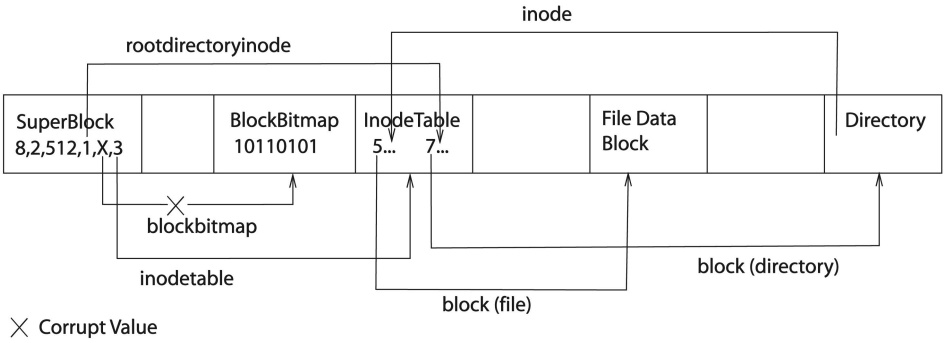


Fig. 2. Inconsistent file system.

be justified even in the absence of any expectation that it will deliver correct execution. If, on the other hand, it is safe for the system to terminate and either external intervention is available or it is preferable to do without the service that the system provides rather than risk the potential consequences of incorrect execution, termination in the face of data structure corruption may be more appropriate.

### 1.4 Consequences of Faults, Errors, and Failures

It is important to separate the concepts of correctness and acceptability—in many situations a system with faults, errors, and even failures may very well be acceptable to its users. A system may be acceptable, for example, as long as the time between failures is sufficiently long or the number of failures in a given time frame is sufficiently small [22]. Or users may accept a system with failures if the cost or severity of the failures is small enough [52]. In practice, given the difficulty of delivering failure-free systems, the important distinctions and concerns often focus not on ensuring correctness, but on trade-offs between the engineering effort required to discover and eliminate faults and the consequences of leaving faults in place. Most software development organizations acknowledge this reality by prioritizing known faults, in many cases choosing to release a system with known low-priority or low-cost failures rather than investing the engineering effort required to eliminate the corresponding faults. Data structure repair may be useful in such efforts if it can convert high-priority or high-cost failures (such as a system crash) into low-priority or low-cost failures (such as continued execution with some anomalies). Of course, in some situations, it may even eliminate the failure completely.

Finally, we note that in some domains (such as scientific computing), it may be more productive to evaluate software from the perspective of accuracy rather than correctness. The key issue is not the presence or absence of faults, but the accuracy of the result that the program produces. Indeed, the program may (for reasons such as floating point rounding errors) be inherently incapable of producing a result that is completely accurate. In some situations, it is possible to exploit the additional flexibility that an accuracy-based perspective provides to trade off acceptable accuracy losses in return for other benefits, such as increased performance or fault tolerance [43].

### 1.5 Contributions

This paper makes the following contributions:

- **Basic Repair Approach.** It presents an approach that uses an abstract model to express important data structure consistency properties. Violations of these properties are repaired by automatically translating abstract model repairs back through the model definition rules to automatically derive a set of concrete data structure updates that implement the repair.
- **Repair Translation.** It presents an algorithm that uses goal-directed reasoning to derive a set of data structure updates that implement the repair.
- **Repair Dependence Graph.** It introduces the repair dependence graph, which captures dependences between consistency constraints, repair actions, and the abstract model. This graph supports formal reasoning about the effect of repairs on both the model and the data structures. It also presents a set of conditions on the repair dependence graph. These conditions forbid certain kinds of cycles that would cause the repair algorithm to loop forever. It also presents an algorithm that removes nodes in the graph to eliminate problematic cycles. These removals prevent the repair algorithm from choosing repair strategies that may not terminate.
- **Experience.** It presents our experience using data structure repair on several applications. Our experience indicates that data structure repair enables our applications to successfully recover from data structure corruption errors.

## 2 File System Example

We next present a simple file system example that illustrates the operation of our repair algorithm. Fig. 2 presents a graphical representation of the file system in our example. The file system consists of an array of disk blocks. In order to quickly allocate new blocks, the file system keeps a table of which blocks are in use. The file system reserves a block, the bitmap block, for storing this table. The file system reserves another block to store the inode table, which keeps track of which blocks store the information for a particular directory or a file. Finally, the first block in the file system is reserved for the superblock, which is a special

```
structure Disk {
     // Array of Blocks in the Disk
     Block  b[d.s.numberofblocks];
     // First block is a superblock
     label b[0]: SuperBlock s;
}
structure Block {
     // Block contains d.s.blocksize bytes
     reserved byte[d.s.blocksize];
}
structure SuperBlock subtype of Block {
     // Number of blocks in file system
     int numberofblocks;
     // Number of inodes in file system
     int numberofinodes;
     // Size of blocks in bytes
     int blocksize;
     // Inode of Directory
     int rootdirectoryinode;
     // Block where Block Bitmap resides
     int blockbitmap;
     // Block where Inode Table resides
     int inodetable;
}
structure BlockBitmap subtype of Block {
     // Bit array to store block's used status
     bit bitmap[d.s.numberofblocks];
}
structure DirectoryBlock subtype of Block {
     // Directory entries in the directory
     DirectoryEntry de[d.s.blocksize/128];
}
structure DirectoryEntry {
     // Name of file
     byte name[124];
     // Inode that stores file's blocks
     int inode;
}
structure InodeTable subtype of Block {
     // Array of inodes
     Inode itable[d.s.numberofinodes];
}
structure Inode {
     // Array of blocks where data is stored
     int block[12];
     // Reference count for inode
     int referencecount;
}

Disk *d;
```

Fig. 3. Structure definitions.

block that stores critical file system configuration information. For example, it stores the size of the blocks in the file system, which block contains the block bitmap, and which block contains the inode table.

The particular file system shown in Fig. 2 has an inconsistency. As indicated by the "X," the bitmap block location stored in the superblock has been corrupted and no longer stores the correct location of the bitmap block. As a result, future block allocations will fail.

## 2.1 Consistency Specification

The data structure consistency specification consists of two parts: a part that specifies a translation from the concrete data structures into an abstract model and a part that specifies consistency constraints that this abstract model must satisfy. The translation part of the specification consists of the data structure declarations in Fig. 3 (these declarations specify the physical layout of the data

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

Fig. 4. Set and relation declarations.

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

Fig. 5. Model definition rules.

structures that comprise the file system), and the model definition, which consists of the set and relation definitions in Fig. 4 (these definitions specify the sets and relations in the model of the file system) and the model definition rules in Fig. 5 (these rules specify how to construct the abstract model from the data structures).

### 2.1.1 Structure Declaration

The first part of the translation specification says how the data structures are physically laid out in memory. This part of the specification uses a structure definition language that is similar to C structure definitions with a few extensions. One of these extensions allows the developer to specify variable length arrays in which the length is stored in a data structure. We now examine the data structure definitions in Fig. 3 in more detail.

The file system consists of an array of `Block` objects. The first line in the `Disk` structure definition declares that the `Disk` object contains this array of blocks. The second line declares a label `s` for the first block in the file system and that this block has the type `Superblock`. Note also that the size of the array of blocks is given by the expression `d.s.numberofblocks`. The `d` variable in this expression

refers to the Disk *d; declaration at the bottom of the figure. This declaration says that the variable d points to a Disk object and that this value is provided to the repair algorithm from the underlying application.[1] Note that, even though d is declared as a pointer, our specification language uses the notation d.s to access the member s. The s refers to the label s in the Disk type declaration, and the numberofblocks refers to the numberofblocks field in the SuperBlock type declaration.

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

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

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

The DirectoryBlock contains an array of directory entries. Each DirectoryEntry contains the name of a file and a reference to the file's inode.

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

### 2.1.2 Model Definition

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

Fig. 4 presents the set and relation declarations for our abstract model of the file system example. The declaration set AllBlocks of Block : UsedBlock | FreeBlock says that the set AllBlocks contains data structures of type Block and contains two subsets: UsedBlock and FreeBlock. We have omitted the declarations of sets that are already declared as a subset of another set and contain no further subsets. In general, the set declarations in Fig. 4 are of the form set $S$ of $T : S_1 | \ldots | S_n$. Such a declaration specifies that the set $S$ in the model contains objects of type $T$ (these types are either base types such as int or structures) and that the sets $S_1, \ldots, S_n$ are subsets of the set $S$. If a set has no subsets, the declaration has the form set $S$ of $T$. The declaration relation InodeOf: DirectoryEntry X UsedInode says that the relation InodeOf relates objects in the set DirectoryEntry to objects in the set UsedInode. In general, the relation declarations in Fig. 4 are of the form relation $R : S_1 \times S_2$. Such a declaration says that the relation $R$ relates the objects in set $S_1$ to the objects in set $S_2$.

We have discussed how the sets and relations in the abstract model are declared. We next discuss how the developer specifies the translation between the concrete data structures and the sets and relations in the abstract model. Conceptually, the model definition rules in Fig. 5 specify how to traverse the data structures to build the sets and relations in the abstract model. The model definition rules are of the form Quantifiers, Guard => Inclusion Condition. Each rule specifies quantifiers that identify the scope of the variables in the body. The inclusion condition specifies an object (or tuple) that must be in a specific set (or relation) if the guard is true. The repair algorithm evaluates the model definition rules on the concrete data structure to generate the abstract model. Fig. 5 presents the model definition rules for the file system example.

The first model definition rule places the first block in the file system in the SuperBlock set. The next two model definition rules place the d.s.blockbitmap and d.s.inodetable elements of the block array into the BlockBitmap and InodeTable sets, respectively. The as keyword in these two model definition rules tells the compiler to view the Block data structure as a BlockBitmap or InodeTable data structure, both of which structurally inherit from the Block structure definition. These three model definition rules identify key blocks in the file system and place them into sets.

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

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

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

1. In practice, the repair algorithm requires primitive actions to read and write the disk blocks. Our implementation uses the memory mapping facility of the UNIX operating system to map the disk data structure into memory. Then, the repair algorithm can read and write to this structure as it would to any other data structure in memory.

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

Fig. 6. Consistency constraints.

In general, we intend that developers will use the sets in the abstract model to group together all the objects with the same consistency properties. For example, the 11th model definition rule places all of the blocks that store the contents of files in the `FileBlock` set. We intend that developers will use relations to map these objects to primitive values (or other objects) with which these objects are conceptually associated.

### 2.1.3 Consistency Constraints

Consistency constraints specify the data structure consistency properties that should hold for the abstract model. Consistency constraints are specified using the consistency constraint language. The specification language allows the developer to use the logical connectives (and, or, not) to assemble the body of a constraint out of atomic propositions. These atomic propositions express basic properties on the sets and relations. Our consistency constraint language includes universal quantifiers that can quantify over the objects in the sets or the tuples in the relations. The consistency constraints in Fig. 6 identify the consistency properties that the file system model must satisfy.

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

The next constraints specify properties that all objects in a given set must satisfy. The third constraint ensures that the `BlockStatus` relation maps blocks in the `UsedBlock` set to the Boolean value `true`, and the fourth consistency constraint ensures that the `BlockStatus` relation maps blocks in the `FreeBlock` set to the Boolean value `false`. The combined effect of these two constraints is to ensure that the `BlockStatus` relation correctly records whether blocks are in use or free. Note that these constraints use the `BlockStatus` relation as a function. Our system allows such uses, provided either that the compiler can determine that such a relation is a function by construction or a second constraint ensures that the relation is a function.

The fifth consistency constraint specifies that the reference count for each used inode must reflect the number of directory entries that refer to that inode.[2] The final consistency constraint ensures that each file or directory block is referenced by at most one directory entry. Formally, this constraint ensures that the inverse of the `Contents` relation evaluated on a member of the `FileDirectoryBlock` set contains exactly one object. In general, each consistency constraint is a first-order logical formula consisting of a sequence of quantifiers followed by a quantifier-free Boolean formula of atomic propositions.

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

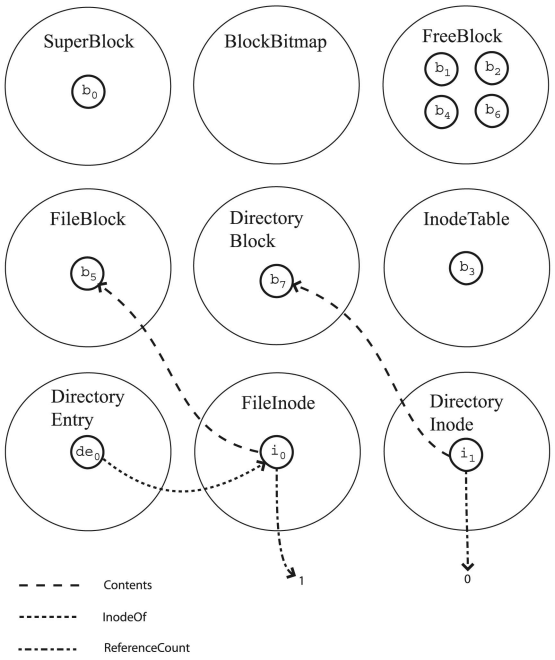


Fig. 7. Broken model.

## 2.2 Repair Algorithm

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

We illustrate the operation of the repair algorithm by discussing the steps it takes to repair a file system whose `Superblock` has an out-of-bounds bitmap block index. At the end of the model construction process, the repair algorithm constructs the abstract model shown in Fig. 7.

Notice that the `BlockBitmap` set in Fig. 7 is empty. This occurs because the bitmap block index `d.s.blockbitmap` is out-of-bounds, and therefore the second model definition rule does not insert any blocks into the `BlockBitmap` set. Since the `BlockBitmap` set is empty, the model violates the first consistency constraint from Fig. 6. To repair this violation, the repair algorithm performs a model repair that adds a block from the `FreeBlock` set (the developer specifies this set as the source of new `Blocks` to insert into other sets as described in Section 4) to the `BlockBitmap` set.

At this point, the repair algorithm must translate this model repair into an update on the concrete data structure. It uses goal-directed reasoning to perform this translation as follows. To generate an update that implements this addition, the compiler finds the model definition rule that

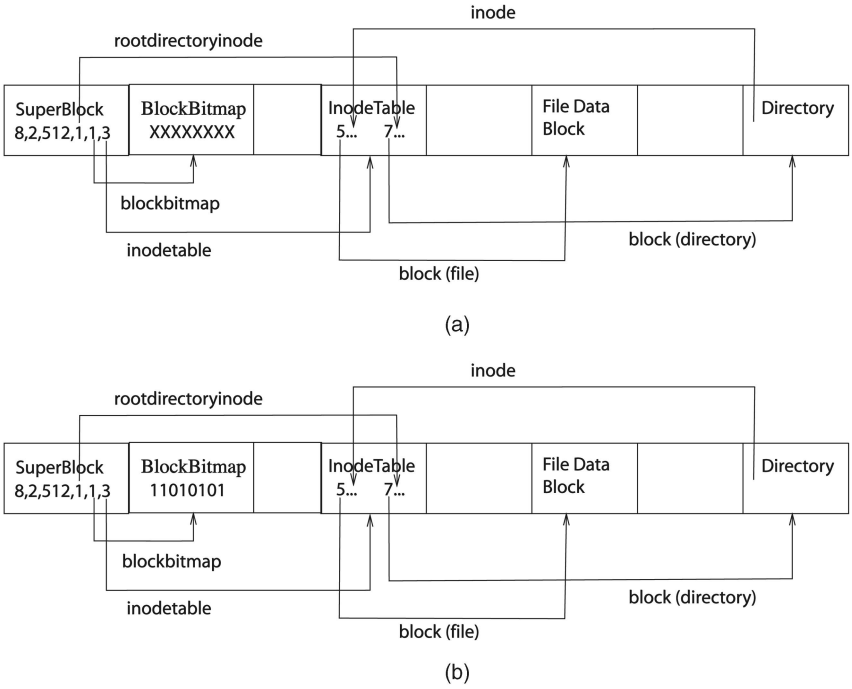


Fig. 8. Repair sequence. (a) Allocated block bitmap. (b) Repaired file system.

constructs the `BlockBitmap` set. The relevant model definition rule from Fig. 5 is `true => d.b[d.s.blockbitmap] as BlockBitmap in BlockBitmap`. The compiler analyzes this model definition rule to determine that the inclusion condition of this rule adds the block from the array `d.b` at offset `d.s.blockbitmap` to the `BlockBitmap` set. As a result, the repair algorithm can make this model definition rule add a block to the `BlockBitmap` set by calculating the block's index in `d.b` and setting `d.s.blockbitmap` equal to this value. Notice that calculating the block's index in `d.b` only works if the selected block is a member of the array `d.b`. To ensure that the selected block is a member of the array `d.b`, it suffices to show that all members of the set from which the block was selected, the `FreeBlock` set, are in this array. To check this condition, the compiler analyzes the rule that constructs the `FreeBlock` set to determine that all blocks in the `FreeBlock` set are from the array `d.b`, and therefore it can set `d.s.blockbitmap` to the index `j` of the block `d.b[j]`. The resulting file system is shown in Fig. 8a. An additional effect of the data structure update is that the block becomes a member of the set `UsedBlock` of used blocks and is removed from the `FreeBlock` set.

After this update, the repair algorithm rebuilds the abstract model. This rebuilt abstract model is given in Fig. 9. Notice that, although the `BlockStatus` relation now maps blocks to Boolean values, it does not correctly map blocks in the `FreeBlock` set to the value `0` (representing false) nor blocks in the `UsedBlock` set to the value `1` (representing true). Therefore, when the repair algorithm checks the consistency constraints, it discovers several violations of the consistency constraints 3 and 4 in Fig. 6. These violations occur because the bits in the new bitmap block do not correctly reflect which blocks are free and which blocks are in use. The repair algorithm repairs each of these violations by repairing the incorrect tuples in the `BlockStatus` relation to reflect the contents of the `UsedBlock` and `FreeBlock` sets—if a block $u$ is used, the repair algorithm ensures that $\langle u, true\rangle$ (and no other tuple with $u$ as its first component) is in the `BlockStatus` relation, and if a block $u$ is free, the repair algorithm ensures that $\langle u, false\rangle$ is in the `BlockStatus` relation.

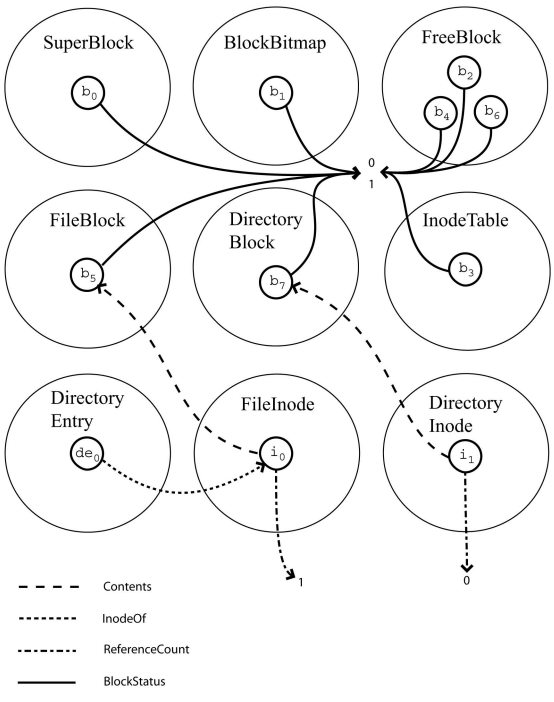


Fig. 9. Model with a BlockBitmap.

As before, the compiler next finds the model definition rule that constructs the BlockStatus relation. The relevant model definition rule from Fig. 5 is

```
for j = 0 to d.s.numberofblocks − 1,
for bbb in BlockBitmap,
true => < d.b[j], bbb.bitmap[j] > in BlockStatus.
```

Notice that this model definition rule contains the two quantified variables j and bbb. To use this model definition rule to modify the tuples in the BlockStatus relation, the repair algorithm must determine the appropriate objects to bind to these quantified variables. Since bbb quantifies over the singleton set BlockBitmap, the compiler trivially determines that the variable bbb must be bound to the only object in the BlockBitmap set. However, it must still determine what integer to bind to the quantified variable j. It is possible to resolve the quantified variable bindings by matching the expression in the inclusion condition to the object (or tuple) to be added or removed to or from the set (or relation). The generated repair code uses this strategy to generate data structure updates for model repairs that add objects (or tuples) to sets (or relations).

However, if the model repair removes an object (or tuple) from a set or relation or if the model repair modifies a preexisting tuple, the compiler can use a simpler strategy. In these cases, it is not necessary to match the expressions in the inclusion condition to the object (or tuple) to be removed. Instead, the generated repair code can simply rebuild the model to discover the quantifier bindings that cause the model definition rule to add the object (or tuple) to the set (or relation). It then uses these discovered bindings to perform a data structure update that removes the object (or tuple) or modifies the tuple. In effect, the repair algorithm uses a lazy repair generation algorithm that delays the application of the data structure update until the next model reconstruction.

For example, to repair an incorrect tuple in the BlockStatus relation, the generated repair algorithm starts rebuilding the model. Whenever model definition rule 6 in Fig. 5 attempts to add the incorrect tuple to the BlockStatus relation, the repair algorithm stops the model construction to repair this incorrect tuple. At this point, the repair algorithm has the quantified variable bindings that cause model definition rule 6 to add the incorrect tuple and can use these variable bindings to perform a data structure update that repairs the incorrect tuple by setting the element (bbb.bitmap[j]) in the concrete data structure to the value specified by the model repair (0 for blocks in the FreeBlock set and 1 for blocks in the UsedBlock set). Part B of Fig. 8 presents the repaired file system after these updates have been performed.

Fig. 10 shows the rebuilt abstract model. The repair algorithm checks the consistency constraints on this model and finds that the repaired model satisfies all of the consistency constraints. Therefore, the repair process is complete and the repair algorithm exits.

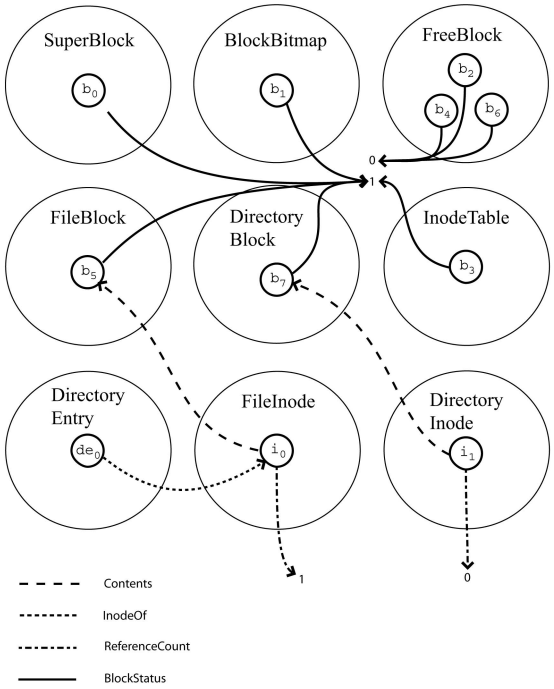


Fig. 10. Repaired model.

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

### 2.3 Repair Dependence Graph

A basic issue in ensuring repair termination is that repairing one constraint may cause the repair algorithm to violate another constraint. If the repair of the newly violated constraint, in turn, causes the originally repaired constraint to become violated, there is an infinite repair loop. The compiler uses a repair dependence graph to reason about termination (see Section 5.3). The edges in this graph capture any invalidation effects that the repair of one constraint may have on other constraints; the absence of cycles in this graph guarantees that all repairs will terminate.

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

## 3 Overview of Repair Algorithm

At this point, we have presented a file system example that illustrates the operation of the repair algorithm. We next present the repair algorithm in more detail. Our compiler generates repair algorithms that use the following basic repair strategy:

1. **Initial Model Construction.** The repair algorithm constructs an abstract model.
2. **Inconsistency Detection.** The repair algorithm evaluates the consistency constraints. If the repair algorithm finds a violation, it proceeds to the next step. Otherwise, the data structure is consistent and the repair process exits.
3. **Conjunction Selection.** The repair algorithm selects one of the conjunctions in the disjunctive normal form of the violated constraint. It will ensure that the constraint holds by repairing the atomic propositions in this conjunction. The conjunction choice can be controlled by the developer or by a cost function that assigns a cost to the repair of each atomic proposition.
4. **Model Repair.** For each violated atomic proposition in the conjunction, the repair algorithm performs an abstract repair on the model. These model repairs either add or remove objects (or tuples) from sets (or relations) to satisfy the violated atomic propositions.
5. **Data Structure Updates.** The repair algorithm must perform data structure updates to implement the model repair. If the repair algorithm performs a model repair that adds an object (or tuple) to a set (or relation), it immediately performs the corresponding data structure update. If an update removes an object (or tuple) from a set (or relation) or atomically modifies a relation, the repair algorithm records that this data structure update should be performed when the model is rebuilt. Delaying these data structure updates enables the specification compiler to generate data structure updates without knowing the specific quantifier bindings for the model definition rules. Step 6 performs any delayed data structure updates as it rebuilds the model.
6. **Model Update.** The repair algorithm performs the model construction described in Step 1. Whenever an object (or tuple) is added to a set (or relation), the repair algorithm checks if the object (or tuple) was in the set (or relation) in the previous version of the model from Step 4. If the object (or tuple) was not in the set (or relation), the repair algorithm checks if a specific data structure update has been recorded for the given object (or tuple) and set (or relation). If one has, the repair algorithm performs that data structure update as described in Step 5. Otherwise, it checks if a compensation update exists for the rule responsible for the addition of the new object (or tuple). Compensation updates are used to make data structure updates more precisely implement the corresponding model repair and are describe in more detail in Section 3.3.4. If one exists, the repair algorithm performs the compensation update in the same manner as Step 5. If the repair algorithm performs any updates, it recomputes the model. Once the repair algorithm completes this recomputation, it deletes the old model and deletes the updates recorded for objects or tuples. Then, it proceeds to Step 2.

The remainder of this section discusses this algorithm in more detail. Section 3.1 discusses Step 1, Section 3.2 discusses Step 2, and Section 3.3 discusses Steps 3 through 6.

### 3.1 Model Construction

The model definition rules specify a translation from the concrete data structures to an abstract model. The model construction phase constructs the abstract model by evaluating the model definition rules applied to the concrete data structure.

In our experience, model definition rules sometimes contain expressions that do not depend on the quantified variables. A naive implementation would reevaluate these expressions for each quantifier binding. Our specification compiler implements the standard loop invariant hoisting optimization. When the specification compiler determines that an expression does not depend on a quantified variable, it lifts the evaluation of that expression outside of the given quantifier evaluation. For example, this algorithm would hoist the evaluation of `map.xsize` outside of the quantifiers in the model definition rule

```
for t in GRID, for x = 0 to map.xsize,
for y = 0 to map.ysize - 1,
true => t.grid[x + (y * map.xsize] in TILE.
```

This optimization corresponds to loop-invariant code motion.

We next discuss how we address the two significant complications in this process: the presence of negation in the model definition rules and the possibility that pointers to data structures may be corrupted.

#### 3.1.1 Negation and the Rule Dependence Graph

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

To address this issue, we allow negation only when a model definition rule's negated inclusion constraint does not depend directly on the set or relation constructed by that model definition rule or indirectly on this set or relation through the actions of other model definition rules. For example, this restriction would prevent a developer from writing the model definition rule `for s in S, !(s in T) => s in T`. For a given model definition rule with negated dependences on sets and/or relations, this restriction allows the model construction algorithm to completely construct those sets and/or relations before performing the fixed-point computation that evaluates the

given model definition rule. Because these sets and/or relations are completely constructed, negation of inclusion constraints that reference them does not affect the fixed point computation.

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

#### 3.1.2 Pointers

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

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

We next extend the model construction algorithm to check that each `struct` accessed via a pointer is valid before the model construction algorithm inserts the `struct` into a set or a relation. All valid `structs` reside completely in allocated memory. In addition, if two `structs` overlap, one must be completely contained within the other and the declarations of both `structs` must agree on the format of the overlapping memory. This approach ensures that only valid `structs` appear in the model. If two data structures illegally overlap, the repair algorithm nullifies the reference to one of the data structures. This guarantees that write operations to one data structure will not corrupt the other data structure and that the model construction algorithm is deterministic.

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

### 3.2 Consistency Checking

Once our tool has constructed an abstract model, it executes the consistency checking algorithm on this model. For each model constraint, the checking algorithm iterates through the legal quantifier bindings. The checking algorithm then evaluates the constraints. If the consistency checking algorithm finds a constraint violation, the repair algorithm performs repairs as described in Section 3.3.

### 3.3 Repairing a Single Constraint

The inconsistency detection algorithm iterates over all values of the quantified variables in the consistency constraints, evaluating the body of the constraint for each possible combination of the values. If the body evaluates to false, the algorithm has detected a violation and has computed an explicit set of bindings for the quantified variables that causes the constraint body to evaluate to false. To generate a repair action for a constraint, the compiler converts the constraint to disjunctive normal form (disjunctions of conjunctions of atomic propositions) and then generates code that performs steps 3 through 6 in the repair algorithm description given in the beginning of Section 3. The steps from repair algorithm that are used to repair a single constraint violation are listed below:

1. **Conjunction or Quantifier Selection.** Satisfying all of the atomic propositions in any of the constraint's conjunctions will ensure that the constraint is satisfied. Alternatively, the repair algorithm may remove an object (or tuple) from a set (or relation) that the constraint is quantified over to eliminate the quantifier binding that makes the constraint false. The first step is therefore to either select a conjunction to satisfy or an object (or tuple) to remove from a set (or relation) that the constraint quantifies over.
2. **Model Repair.** Each atomic proposition has a set of model repair actions that, when performed, ensure that the atomic proposition is satisfied. The next step is therefore to perform these repair actions.
3. **Data Structure Updates.** The repair algorithm uses a set of data structure updates to implement each model repair action; the model definition rules determine the specific set of updates.
4. **Model Update and Compensation Updates.** The repair algorithm rebuilds the model as previously described. While rebuilding the model, the repair algorithm performs data structure updates that remove objects (or tuples) from sets (or relations), data structure updates that modify tuples, and any compensation updates.

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

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

#### 3.3.1 Model Repair Actions

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

- **Size Propositions.** For size propositions, such as `size(BlockBitmap)` $= 1$, the generated repair algorithm simply adds or removes the minimal number of objects (or tuples) to or from the appropriate set (or relation) necessary to satisfy the proposition.
- **Inequalities.** For inequality propositions, such as `i.ReferenceCount` $=$ `size(InodeOf.i)`, the generated repair algorithm computes the right-hand side of the inequality, adds or subtracts 1 if the comparison is a greater than or less than comparison, and then assigns this value to the left-hand side of the inequality. For the not equals inequality, the specification compiler currently generates code that adds one to the right-hand side and assigns this value to the left-hand side. Note that the compiler rewrites the comparison operation of a negated inequality to remove the negation.

  In general, inequalities can be solved by modifying the relations that appear on the right-hand side. Our specification compiler does not currently generate such repair actions. However, the user can always rewrite the constraint so that these relations appear on the left-hand side instead.
- **Inclusion Propositions.** To make an inclusion proposition true, the generated repair algorithm adds the specified object (or tuple) to the specified set (or relation). To make an inclusion proposition false, the generated repair algorithm removes the specified object (or tuple) from the specified set (or relation).

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

#### 3.3.2 Updates that Add Objects or Tuples

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

We normalize the guards to disjunctive normal form. For each combination of model definition rule that may cause an object (or tuple) to be added to a set (or relation) and conjunction in the DNF form the rule's guard, the compiler matches the inclusion condition in the model definition rule to the object (or tuple) to be added, then generates code that performs updates to the data structure to ensure that all of the propositions in the conjunction are true and that the model definition rule's inclusion condition is equal to the object (or tuple) to be added. The specific update depends on the form of the proposition; e.g., for inequality propositions such as `v.f` $< E$, the update computes $E$, subtracts 1 from $E$ to generate a value that satisfies the proposition, then assigns this value to `v.f`.

To generate code that performs an update that adds a new object (or tuple) to a set (or relation) using a model definition rule, the compiler needs to know what objects the quantified variables should be bound to. The compiler resolves these bindings using one of two different strategies:

- If a set (or relation) that the model definition rules quantifies over contains at most one object (or tuple), then the corresponding variable binding is trivially equal to the only object (or tuple) in that set (or relation).[3]
- The compiler can match expressions in the inclusion condition of the model definition rule to the object (or the appropriate half of the tuple) to be added. This generates equations that may specify a quantified variable in terms of the object (or tuple) to be added.[4] For example, consider the task of using the model definition rule `for n in N, n.x > 0 =>` $<$ `n, n.x >  in R` to generate a data structure update that adds the tuple $\langle o, 3 \rangle$ to `R`. The compiler would match the expressions in the inclusion condition of this rule to the members of the tuple to be added to generate the equations `n` $= o$ and `n.x` $= 3$. Note that the first equation provides the binding for the variable `n` in terms of the tuple to be added.

Finally, it is possible for one operation in an update to change state that is referenced by another operation in the update. In principle, this interference could cause an update to invalidate the change performed by another update. The specification compiler handles this issue by constructing a dependence graph between the various operations, then topologically sorting this graph. If the graph contains cyclic dependences, the specification compiler rejects the update. Otherwise, the compiler generates code that performs the operations in the update in the topological sort order. This order ensures that the updated state correctly contains the effects of all of the operations.

#### 3.3.3 Updates that Remove Objects or Tuples

The compiler uses a similar strategy to implement repairs that remove an object (or tuple) from a set (or relation), with one major simplification: It is not necessary to match the expressions in the inclusion condition to the object (or tuple) to be removed or the tuple to be modified. Instead, the generated repair code can simply rebuild the model to discover the quantifier bindings that cause the model

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

4. Note that the compiler may need to generate additional model repairs that ensure that the objects are included in the sets that the model definition rule quantifies over.

definition rule to add the object (or tuple) to the set (or relation). It then uses these discovered bindings to perform a data structure update that removes the object (or tuple) or modifies the tuple. In effect, the repair algorithm uses a lazy repair generation algorithm that delays the application of the data structure update until the next model reconstruction.

To generate a data structure update that removes an object (or tuple), the specification compiler chooses a set of propositions that includes at least one proposition from each conjunction of each rule that could cause the object (or tuple) to appear in the set (or relation). It then generates actions that falsify the propositions in this set. Finally, the compiler checks that there is no dependence cycle between propositions that use and define the same field or variable. The compiler generates a data structure update that satisfies the corresponding set of propositions in a dependence-preserving order.

#### 3.3.4 Compensation Updates

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

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

## 4 Developer Control of Repairs

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

### 4.1 Controlling the Repair Actions

The developer can specify that the repair algorithm should not modify certain fields, sets, or relations. The repair algorithm can then provide feedback that characterizes the inconsistencies that can be repaired without modifying these elements. The developer can provide hand-coded routines to repair certain consistency violations.

### 4.2 New Objects

A repair action may need a source of new objects to add to sets to bring them up to the specified size. As illustrated in Section 2, other sets (as specified in the set and relation definition) are one potential source. For primitive types, such as integers, the action can simply synthesize new values. For structs, memory allocation primitives are a potential source of new objects. The developer can specify the source of the object; a typical source is a memory allocator or another set of objects. We similarly allow the developer to control the source of tuples added to relations. In the absence of such guidance, the compiler uses heuristics to choose a source.

### 4.3 Invoking Check and Repair

Our 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 simply to 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 consistent 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.

## 5 The Repair Dependence Graph

We have presented our core repair algorithm. However, we have not yet discussed how our specification compiler determines that a generated repair algorithm terminates. The specification compiler constructs a *repair dependence graph* $\langle N, E \rangle$ to reason about the termination of the repair algorithm. Nodes in this graph represent conjunctions in the DNF of the consistency constraints, repair actions, and model definition rules. One node has a dependence on a second node if the repair algorithm may be required to performed the event represented by the first node as a result of the event represented by the second node, or if the event represented by the first node may occurs as a result of the event represented by the second node. For example, we say that a data structure update depends on the corresponding model repair, because the repair algorithm may have to perform the data structure update to implement the model repair. Note that events in an individual repair

follow paths on the graph. For example, the repair algorithm decides to repair a conjunction; then it performs a model repair; then it implements this model repair by performing the corresponding data structure update, which may change the scope of model definition rules; and finally the repair algorithm may perform compensation updates to counteract these scope changes. This chain of events corresponds to the paths on the repair dependence graph that start from the model conjunction.

Edges capture dependences between the consistency constraints, repair actions, model definition rules, and choices in the repair process. In particular, an edge may denote that the repair of a constraint requires a given model repair, that the implementation of a model repair requires a given data structure update, that performing a repair action may affect what objects (or tuples) model definition rules add to sets (or relations), or that a repair action or change in a model definition rule's scope may violate a constraint. The absence of cycles in the repair dependence graph ensures that the corresponding repair algorithm will not perform any infinite repair sequences and therefore terminates.

### 5.1 Nodes in Role Dependence Graph

The graph contains the following nodes:

- **Model conjunction nodes.** In disjunctive normal form, each consistency constraint $C_i$ is of the form

$$C_i = Q_{i1}, \ldots, Q_{im} \bigvee_{j}^{j_{max}} C_{ij},$$

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

### 5.2 Edges in the Graph

This section provides only an overview of how the specification compiler generates the edges in the repair dependence graph. More details are provided in Demsky's thesis [15].

The edges $E$ in the repair dependence graph represent how the model and data structure repairs may affect other parts of the model and data structures. The important dependence chains flow

1. from repaired conjunctions to conjunctions that the repairs may falsify,
2. from repaired conjunctions to quantifiers whose scope the repair may increase or decrease,
3. from data structure updates to conjunctions that the update may falsify, and
4. from data structure updates to quantifiers whose scope the repair may increase or decrease.

For example, there is an edge $\langle N_{ij}, A_{ijkl} \rangle \in E$ from each model conjunction node $N_{ij}$ to each abstract repair node $A_{ijkl}$ that may repair one of the atomic propositions in the conjunction. There are other edges to capture dependences between each of the different classes of nodes. The graph contains edges to model the following dependences:

#### 5.2.1 Model Repair Effects

There must be an edge from a model repair node to a conjunction node if the model repair may falsify the conjunction. The compiler uses a procedure that determines if the repair of a first atomic proposition may falsify a second atomic proposition (this proposition is taken from the conjunction that the repair of the first proposition may falsify).

#### 5.2.2 Data Structure and Compensation Updates

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

#### 5.2.3 Scope Increases and Decreases

Increases or decreases in the scope of a model definition rule may change the abstract model. In particular, if the change in scope of a model definition rule causes an object

(or tuple) to be added to or removed from a set (or relation), the resulting change in the model may falsify consistency constraints that depend on the set (or relation) or cause additional changes in the scopes of other model definition rules. The repair dependence graph contains edges that account for these possibilities.

### 5.3 Termination

By construction, the edges in the graph capture all of the repair dependences of the repair algorithm. As a result, the transitive closure of the edges from a conjunction node captures all of the possible effects of repairing that model conjunction. Any infinite repair sequence therefore shows up as a cycle.

The repair dependence graph must be acyclic with the exception of cycles that contain scope decrease and consequence nodes only, cycles that contain scope increase and consequence nodes only, or cycles that are not reachable from the model conjunction nodes. Note that these cycles do not affect termination as scope decrease cycles have no work associated with them, scope increase cycles can only discover as many objects as exist in the heap, and the actions in unreachable cycles are never used. The repair algorithm generator may remove model conjunction nodes, data structure update nodes, and consequence/compensation update nodes to satisfy these cyclity constraints. The generated repair algorithm never performs the repair actions that correspond to the deleted nodes. The final graph must satisfy the following conditions in order to ensure that repairs exist for violated constraints: 1) There is at least one model conjunction node for each constraint in the model, 2) each abstract repair node has at least one edge to a data structure update, and 3) each scope increase or decrease node has at least one edge to a consequence or compensation update node.

## 6 Experience

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

### 6.1 Methodology

We implemented our data structure repair algorithm. This implementation consists of approximately 20,800 lines of Java code and C code; the implementation compiles specifications into C code that performs the consistency checks and (if necessary) repairs the data structures. The source code for the tool and sample specifications are available at http://www.cag.lcs.mit.edu/~bdemsky/repair. We ran the applications (with the exception of the parallel x86 emulator) on an IBM ThinkPad X23 with an 866 MHz Pentium III processor, 384 MB of RAM, and RedHat Linux 8.0.

For each application, we identified important consistency constraints and developed a specification that captured these constraints. We also obtained a workload that caused the application to generate corrupt data structures. When possible, the workload triggered a known programming error. In other cases, we used fault insertion to mimic either the effect of a previously corrected programming error or a common data structure inconsistency source. We then compared the results of running a chosen workload with and without inconsistency detection and repair.

### 6.2 AbiWord

AbiWord is a full-featured word processing program available at www.abisource.com. It consists of over 360,000 lines of C++ code and can import and export many file formats, including Microsoft Word documents. It uses a piece table data structure to internally represent documents. The piece table contains a doubly linked list of the document fragments. A consistent piece table contains a reference to both the head and the tail of the doubly linked list of document fragments. A consistent fragment contains a reference to the next fragment in the list and a reference to the previous fragment in the list. Furthermore, a consistent list of fragments contains both a section fragment and a paragraph fragment. We developed a specification for the piece table data structure. Our specification consists of 94 lines, of which 70 contain structure definitions.[5]

A bug in version 0.9.5 (and all previous versions) of AbiWord causes AbiWord to attempt to append text to a piece table which lacks a section fragment or a paragraph fragment. This bug is triggered by importing certain valid Microsoft Word documents, causing AbiWord to fail with a segmentation violation when the user attempts to load the document. We obtained such a document and used our system to enhance AbiWord with data structure repair as described in this paper. Our experimental results show that data structure repair enables AbiWord to successfully open and manipulate the document. Further inspection reveals that loading this document causes AbiWord to attempt to append text to an (inconsistent) empty fragment list. Our repair algorithm detects the attempt to append text to the empty list and repairs the inconsistency by adding a section fragment and a paragraph fragment, breaking any cycles in the fragment list, connecting the fragments using their `next` fields, pointing the `prev` field of each fragment to the previous fragment, and redirecting the `head` pointer to the beginning of the list and the `tail` pointer to the end of the list. The result of this repair is that AbiWord is able to successfully append the text to the list and continue on to read and edit Word documents without the loss of any information. Without repair, AbiWord fails as it attempts to read in the document.

### 6.3 Parallel x86 Emulator

The parallel x86 emulator is a software-based x86 emulator that runs x86 binaries on the MIT RAW machine [50]. The x86 emulator uses a tree data structure to cache translations of the x86 code. To efficiently manage the size of the cache,

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

the emulator maintains a variable that stores the current size of the cache. A bug in the tree insertion method, however, causes (under some conditions) the cache management code to add the size of the inserted cache item to this variable twice. When this item is removed, its size is subtracted only once. The net result of inserting and removing such an item is that the computed size of the cache becomes increasingly larger than the actual size of the cache. The end result is that the emulator eventually crashes when it attempts to remove items from an empty cache.

We developed a specification that ensures that the computed size of the cache is correct. Our specification consists of 110 lines, of which 90 contain structure definitions. Our test workload ran gzip on the x86 emulator. Without repair, the emulator stops with a failed assertion. With repair, the emulator successfully executes gzip.

### 6.4 CTAS

The Center-TRACON Automation System (CTAS) is a set of air-traffic control tools developed at the NASA Ames research center [1]. The system is designed to help air traffic controllers visualize and manage complex air traffic flows. The current source code consists of more than 1 million lines of C and C++ code. Versions of this source code are deployed in the continental United States and are in daily use. CTAS maintains data structures that store aircraft data. Our experiments focus on the objects that store the flight plans. These flight plan objects contain both an origin and destination airport identifier. The software uses these identifiers as indices into an array of airport data structures. Flight plans are transmitted to CTAS as a long character string. The structure of this string is somewhat complicated, and parsing the flight plan string is a challenging activity.

Our fault insertion methodology attempts to mimic errors in the flight plan processing that produce illegal values in the flight plan data structures. When the program uses these illegal values to access the array of airport data, the array access is out of bounds, which typically leads to the program failing because of an addressing error. Our specification captures the constraint that the flight plan indices must be within the bounds of the airport data array. The specification itself consists of 101 lines, of which 84 lines contain structure definitions. The primary challenge in developing this specification was reverse engineering the source to develop an understanding of the data structures. Once we understood the data structures, developing the specification was straightforward.

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

Rebooting CTAS after a crash is an inadequate solution. After a reboot, CTAS takes several minutes to reacquire flight plans and radar data. Furthermore, there are many classes of errors which rebooting does not solve: The system will reacquire the data, reprocess it, and fail again for the same reason. In particular, CTAS will fail whenever it reacquires and attempts to process the faulty flight plan.

### 6.5 Freeciv

Freeciv is an interactive, multiplayer game available at www.freeciv.org. The Freeciv server maintains a map of the game world. Each tile in this map has a terrain value chosen from a set of legal terrain values. Additionally, cities may be placed on the tiles. Our fault injection strategy changes the terrain values in pseudorandomly selected tiles 35 times during the execution of the program. There are two possible errors: illegal terrain values or cities located on an ocean tile instead of a land tile. Our repair algorithm repairs these kinds of errors by assigning a legal terrain value to any tile with an illegal value and by assigning a land terrain value to any ocean tiles containing a city. The specification consists of 191 lines, of which 173 lines contain structure definitions. The principal challenge in developing this specification was reverse engineering the Freeciv source (which consists of 73,000 lines of C code) to develop an understanding of the data structures. Once we understood the data structures, developing the specification was straightforward.

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

### 6.6 A Linux File System

Our Linux file system application implements a simplified version of the Linux ext2 file system. This file system is similar to the one presented in the example, but includes more aspects of the Linux ext2 file system and more consistency properties. The file system, like other Unix file systems, contains bitmaps that identify free and used disk blocks. The file system uses these disk blocks to support fast disk block and inode allocation operations. For our experiments we used a file system with 1,024 disk blocks. Our consistency specification contains 108 lines, of which 55 lines contain structure definitions. Because the structure of such file systems is widely documented in the literature, it was relatively easy for us to develop the specification. In general, we have found that developing specifications is a straightforward task once one understands the relevant data structures.

Our fault insertion mechanism for this application simulates the effect of a system crash: It shuts down the file system (potentially in the middle of an operation that requires several disk writes), then discards the cached state. Our workload opens and writes several files, closes the files, then reopens the files to verify that the data was written

TABLE 1
Time to Check Consistency and Perform Repairs

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

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 then verifies the writes by reading the contents of the files and comparing them to the previous writes.

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.

### 6.7 Performance

To evaluate the performance of our consistency check and repair algorithm, we computed two numbers: 1) the mean time required to perform a consistency check for a consistent data structure, and 2) the mean time required to perform the consistency check and the repair for an inconsistent data structure (rendered inconsistent via fault injection). Table 1 presents the mean consistency check times (over 10 trials) for the different applications and the mean consistency check and repair times. In general, the check and repair times are dominated by the model construction overhead. The check and repair times therefore correlate with the number of times the repair process rebuilds the model. For AbiWord, the mean number of times that the repair algorithm rebuilds the model is 10, for CTAS the mean is 3, for the file system the mean is 119.2, while for Freeciv the mean is 4.6. The number of times the model is rebuilt is, in turn, correlated with the number of data structure updates that the repair algorithm performs. The mean number of updates is 7 for AbiWord, 1 for CTAS, 59.1 for the file system, and 1.8 for Freeciv. As these numbers show, the fault injection strategy for the file system produces faults that require substantially more data structure updates to repair.

### 6.8 Discussion

Our experience indicates that data structure repair can enable a large class of applications to recover from otherwise fatal data structure inconsistencies. However, there are applications for which data structure repair may be undesirable. For example, data structure repair is likely to be inappropriate for numerical calculations that must be absolutely correct and for which the results are not urgently needed. The following factors determine whether or not data structure repair may be appropriate:

- **High cost of halting.** The alternative to data structure repair is typically to halt the system. For many applications, halting can result in large financial losses or even the loss of human life.
- **Lack of better options.** Some systems can recover from failures by rebooting or by falling back on a backup implementation. When these options exist, they may be preferable to data structure repair. However, in some circumstances an error in persistent state or a repeatedly activated fault may cause the system to always crash during or immediately after a reboot. Backup implementations can be prohibitively expensive to develop, and may suffer from the same defects as the primary implementation. In these situations, data structure repair may be the most desirable option.
- **Acceptable repair actions.** In our benchmark applications, the generated repair actions all resulted in acceptable changes to the data structures. Other repair actions may not be acceptable: for example, repairing a file system by reformatting the disk is likely to be unacceptable to most users.
- **Humans in the loop.** Even if a repair action may seem unacceptable (routing a plane to a different airport), the presence of humans in the loop may make the repair action acceptable. Often, human operators may be able to correct small errors in the data (especially if the system provides automated support to locate possibly incorrect data), but may be unable to function without the software system.
- **Well understood consistency properties.** Our benchmark applications all had easily understood consistency properties. It may difficult to develop consistency specifications for legacy applications that manipulate poorly understood data structures.
- **Simple consistency properties.** Our system is unable to automatically generate repair algorithms for some consistency properties. For example, if the developer specifies a complex system of equations that must be satisfied, our compiler will fail to generate a repair algorithm. As a result, the developer will either have to leave out such properties or not use data structure repair.

## 7 Related Work

We survey related work in software error detection [14], [26], traditional error recovery, manual data structure repair, and databases.

Reboot [30], potentially augmented with checkpointing [56], is one approach to error recovery. In the reboot approach, the user simply reboots a crashed or corrupted software system. This returns the system to a known consistent state, the initial state. One drawback of this approach is that all of the volatile state in the software system is lost. Database systems use a combination of

logging and replay to avoid the state loss normally associated with rolling back to a previous checkpoint [23]. There has recently been renewed interest in applying many of these classical techniques in new computational environments such as Internet services [40] and in extending these techniques to reboot a minimal set of components rather than the complete system [10].

Software fault tolerance researchers have developed many methods to address software failures. Recovery blocks [7] allow a developer to provide multiple implementations of a given algorithm and an acceptance test for these implementations. The system executes the first implementation and then performs the acceptance test. If the test passes, the system continues execution. If the test fails, the system repeats the computation using the second implementation followed by the acceptance check. If the acceptance check still fails, the system tries each of the remaining alternate implementations until either the acceptance test passes or the system runs out of implementations in which case it simply fails. This technique requires the developer to expend the effort to develop multiple implementations of a given algorithm and an acceptance test for the recovery block. Furthermore, the recovery block technique may fail if the algorithms share a common defect or if there is an error in the acceptance test. Data structure repair is largely orthogonal to this work. However, our consistency specifications could be used as an acceptance test and the recovery block could fall back to data structure repair instead of aborting.

Backward recovery uses a combination of checkpointing and acceptance testing (or error detection) to prevent a software system from entering an incorrect state [56], [41], [13], [55]. Unfortunately, it can be difficult to handle external actions, such as vending money from an ATM, in this framework. Forward recovery uses multiple copies of a computation to recovery from transient errors [29]. At various points during the execution, the system compares the results between the copies. If a difference is detected, the system starts up another copy of the computation to verify which copy is correct while continuing to execute the copies. After the verification computation completes, the incorrect copies are terminated. Both of these recovery mechanisms are largely orthogonal to data structure repair; they are designed to handle transient faults. These methods cannot recover from deterministic faults as the computation will either fail repeatedly in the case of backward recovery, or all copies of the computation will fail in the same way in the case of forward recovery. Data structure repair may be able to address deterministic faults that corrupt data structures.

In N-version programming, the developer constructs a software system out of multiple, independent implementations and a decision algorithm to decide which result to use in the event of a disagreement [8]. N-version programming can address data structure corruption errors. However, N-version programming may be prohibitively expensive. It requires multiple implementations, which must be independent enough to not share failure modes but similar enough to be comparable. Furthermore, the different versions may still be vulnerable to common mode failures.

Self-checking software is a general term that refers to software that verifies certain aspects of its own correct execution [54]. These aspects include the function of a process, the control sequence of a process, and the data of a process. The software may then take corrective action to recover from detected failures. Data structure repair can be used as a technique to construct self-checking software.

The Lucent 5ESS telephone switch [27], [25], [32], [24] and IBM MVS operating system [39] use inconsistency detection and repair to recover from software failures. Both of these systems contain a set of manually coded procedures that periodically inspect their data structures to find and repair inconsistencies. The reported results indicate an order of magnitude increase in the reliability of the system [23].

Fsck [5], chkdsk [3], and scandisk detect and repair inconsistencies in file systems. These hand-coded applications use domain-specific repairs, such as replicating any blocks that are shared between files. As a result of these domain-specific repair actions, the hand-coded file system repair utilities may preserve more information than our automatically-generated repair algorithms. While our automatically-generated repair algorithms do not currently perform these domain-specific repair actions, some of these repair actions may be general enough that future versions of repair system might include them. Finally, the hand-coded repair algorithms have the potential to be more efficient. The developers of hand-coded repair algorithms may be able to incorporate domain specific optimizations (such as checking certain constraints directly on the data structures) into these repair algorithms. While we have not performed any experiments that compare the performance of our automatically generated repair algorithms to hand-coded repair algorithms, we expect that developers can more easily optimize hand-coded repair algorithms. However, testing and debugging hand-coded repair algorithms may be more challenging—various implementations of file system repair utilities have contained serious errors [4], [2].

Researchers have developed several specific linked data structures, including linked lists and trees, that contain redundancy to enable detecting and repairing errors [47], [48], [49], [46], [33]. One downside of this approach is that the developer must manually design the data structures, develop extra code to maintain the redundant links, and code error detection and recovery routines.

Researchers in the area of self-stabilizing algorithms have developed specific distributed algorithms that eventually converge to a stable state in spite of perturbations [19], [20]. 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.

Researchers have incorporated constraint mechanisms into programming languages. One such system is Kaleidoscope [36]. Kaleidoscope allows the developer to specify

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

Fig. 11. Structure definition language.

| | | |
|---|---|---|
| *M* | := | $(Q,)^*$ *G*=>*I* |
| *Q* | := | for *V* in *S* \| for $\langle V, V\rangle$ in *R* \| for $V = E$ to *E* |
| *G* | := | *G* and *G* \| *G* or *G* \| !*G* \| (*G*) \| *FE*=*E* \| *FE* < *E* \| *FE* <= *E* \| *FE* >= *E* \| *FE* > *E* \| true \| *E* in *S* \| $\langle E, E\rangle$ in *R* |
| *I* | := | *FE* in *S* \| $\langle FE, FE\rangle$ in *R* |
| *E* | := | *FE* \| *number* \| *string* \| *E*+*E* \| *E*-*E* \| *E***E* \| *E*/*E* |
| *FE* | := | *V* \| *FE*.field \| *FE*.field[*E*] |

Fig. 13. Model definition language.

constraints that the system should maintain. The developer is intended to write programs using a hybrid of imperative style programming and constraints where appropriate. Another example of a constraint maintenance system as a programming abstraction is Alphonse [28]. Rule-based programming [38], [35] is a related technique in which the developer defines a test condition and an action to take in response.

Database researchers have developed integrity management systems that enforce database consistency constraints. These systems typically operate at the level of the tuples and relations in the database, not the 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 [12]. This approach has been extended to enable the system to automatically generate both the triggering components and the repair actions [11]. 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) [51]. Our system supports a broader class of constraints—logical formulas instead of Horn clauses. It also supports constraints that relate the value of a field to an expression involving the size of a set or the size of an image of an object under a relation. 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.

Some journaling or log-structured file systems are always consistent on the disk, eliminating the possibility of file system corruption caused by a system crash [44]. Repair remains valuable even for these systems in that it can enable the system to recover from file system corruption caused by other sources such as software errors or hardware damage.

In our previous research, we have developed a specification-based repair system that uses *external constraints* to explicitly translate the model repairs to the concrete data structures [16], [17]. The primary disadvantage of this approach in comparison with the approach presented in this paper is a potential lack of repair effectiveness—there is no guarantee that the external constraints correctly implement the model repairs and, therefore, no guarantee that the concrete data structures will be consistent after repair.

| | | |
|---|---|---|
| *D* | := | set *S* of *T* \| *S* partition $(S,)^*S$ \| *S* subset $(S,)^*S$ \| set *S* of *T* : $(S,)^*S$ \| set *S* of *T* : partition $(S,)^*S$ \| relation *R*:*S*x*S* \| relation *R*:*T*x*T* |

Fig. 12. Set and relation declarations.

## 8 Conclusion

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

## Appendix

See Fig. 11, Fig. 12, Fig. 13, and Fig. 14.

## Acknowledgments

This is a revised and extended version of the paper that appeared in the *Proceedings of the 2005 International*

| | | |
|---|---|---|
| *C* | := | *Q*, *C* \| *B* |
| *Q* | := | for *V* in *S* \| for $\langle V, V\rangle$ in *R* |
| *B* | := | *B* and *B* \| *B* or *B* \| !*B* \| (*B*) \| *P* |
| *P* | := | *VE*=*E* \| *VE*<*E* \| *VE*<=*E* \| *VE*>*E* \| *VE*>=*E* \| *V* in *SE* \| size(*SE*)=*c* \| size(*SE*)>=*c* \| size(*SE*)<=*c* |
| *VE* | := | (*VE*) \| *V*.*R* \| *R*.*V* \| *VE*.*R* \| *R*.*VE* |
| *E* | := | *V* \| *number* \| *string* \| *E*+*E* \| *E*-*E* \| *E****E* \| *E*/*E* \| *E*.*R* \| *R*.*E* \| size(*SE*) \| (*E*) \| sum(*S*.*R*) |
| *SE* | := | *S* \| *VE* |

Fig. 14. Consistency constraint language.

*Conference on Software Engineering* [18]. The authors would like to thank David Wentzlaff for his help with the MIT RAW x86 emulator. They would also like to thank the anonymous referees and the two editors, Dr. William Griswold and Dr. Bashar Nuseibeh, for their criticism and feedback.

## REFERENCES

[1] "Center-Tracon Automation System," http://www.ctas.arc.nasa.gov/, 2006.
[2] "Changelog in Reiserfsprogs Distribution," http://ftp.namesys.com/pub/reiserfsprogs/, 2006.
[3] "Chkdsk," http://www.microsoft.com/resources/documentation/windows/XP/all/proddocs/en-us/chkdsk.mspx?mfr=true, 2006.
[4] "E2fsprogs Release Notes," http://e2fsprogs.sourceforge.net/e2fsprogs-release.html, 2006.
[5] "Ext2 fsck Manual Page," http://e2fsprogs.sourceforge.net/, 2006.
[6] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proc. USENIX Summer Conf.,* 1986.
[7] T. Anderson and R. Kerr, "Recovery Blocks in Action: A System Supporting High Reliability," *Proc. Second Int'l Conf. Software Eng.,* pp. 447-457, 1976.
[8] A. Avizienis, "The Methodology of N-Version Programming," *Software Fault Tolerance,* M.R. Lyu, ed., pp. 23-46, Wiley, 1995.
[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004.
[10] G. Candea and A. Fox, "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel," *Proc. Workshop Hot Topics in Operating Systems (HotOS-VIII),* pp. 110-115, May 2001.
[11] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca, "Automatic Generation of Production Rules for Integrity Maintenance," *ACM Trans. Database Systems,* vol. 19, no. 3, Sept. 1994.
[12] S. Ceri and J. Widom, "Deriving Production Rules for Constraint Maintenance," *Proc. Int'l Conf. Very Large Data Bases,* pp. 566-577, 1990.
[13] K. Chandy and C. Ramamoorthy, "Rollback and Recovery Strategies," *IEEE Trans. Computers,* vol. 21, no. 2, pp. 137-146, Feb. 1972.
[14] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng, "Bandera : Extracting Finite-State Models from Java Source Code," *Proc. 22nd Int'l Conf. Software Eng.,* 2000.
[15] B. Demsky, "Data Structure Repair Using Goal-Directed Reasoning," PhD thesis, Massachusetts Inst. of Technology, Jan. 2006.
[16] B. Demsky and M. Rinard, "Automatic Detection and Repair of Errors in Data Structures," *Proc. 18th Ann. Conf. Object-Oriented Programming Systems, Languages and Applications,* Oct. 2003.
[17] B. Demsky and M. Rinard, "Static Specification Analysis for Termination of Specification-Based Data Structure Repair," *Proc. 14th IEEE Int'l Symp. Software Reliability Eng.,* Nov. 2003.
[18] B. Demsky and M. Rinard, "Data Structure Repair Using Goal-Directed Reasoning," *Proc. 2005 Int'l Conf. Software Eng.,* May 2005.
[19] E.W. Dijkstra, "Self-Stabilization in Spite of Distributed Control," *Comm. ACM,* vol. 17, no. 11, pp. 643-644, 1974.
[20] S. Dolev, *Self-Stabilization.* MIT Press, 2000.
[21] R. Ghiya and L.J. Hendren, "Is It a Tree, a Dag, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C," *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* 1996.
[22] A.L. Goel, "Software Reliability Models: Assumptions, Limitations, and Applicabity," *IEEE Trans. Software Eng.,* vol. 11, no. 12, Dec. 1985.
[23] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
[24] T. Griffin, H. Trickey, and C. Tuckey, "Generating Update Constraints from PRL 5.0 Specifications," *Preliminary Report Presented at AT&T Database Day,* Sept. 1992.
[25] N.K. Gupta, L.J. Jagadeesan, E.E. Koutsofios, and D.M. Weiss, "Auditdraw: Generating Audits the FAST Way," *Proc. 19th Int'l Conf. Software Eng.,* 1997.
[26] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A System and Language for Building System-Specific, Static Analyses," *Proc. SIGPLAN '02 Conf. Program Language Design and Implementation,* 2002.
[27] G. Haugk, F. Lax, R. Royer, and J. Williams, "The 5ESS(TM) Switching System: Maintenance Capabilities," *AT&T Technical J.,* vol. 64, no. 6, part 2, pp. 1385-1416, July-Aug. 1985.
[28] R. Hoover, "Incremental Computation as a Programming Abstraction," *Proc. SIGPLAN '92 Conf. Program Language Design and Implementation,* 1992.
[29] K. Huang, J. Wu, and E.B. Fernandez, "A Generalized Forward Recovery Checkpointing Scheme," *Proc. 1998 Ann. IEEE Workshop Fault-Tolerant Parallel and Distributed Systems,* Apr. 1998.
[30] Y. Huang, C. Kintala, N. Kolettis, and N.D. Fulton, "Software Rejuvenation: Analysis, Module and Applications," *Proc. 25th Int'l Symp. Fault-Tolerant Computing,* 1995.
[31] D. Jackson, "Alloy: A Lightweight Object Modeling Notation," Technical Report 797, Laboratory for Computer Science, Massachusetts Inst. of Technology, 2000.
[32] D.A. Ladd and J.C. Ramming, "Two Application Languages in Software Production," *USENIX 1994 Very High Level Languages Symp. Proc.,* Oct. 1994.
[33] C.-C.J. Li, P. Cheng, and W.K. Fuchs, "Local Concurrent Error Detection and Correction in Data Structures Using Virtual Backpointers," *IEEE Trans. Computers,* vol. 38, no. 11, pp. 1481-1492, Nov. 1989.
[34] J. Lions, "Ariane 5 Flight 501 Failure: Report by the Inquiry Board," 1996.
[35] D. Litman, P.F. Patel-Schneider, and A. Mishra, "Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules," *Proc. 12th Ann. Conf. Object-Oriented Programming Systems, Languages and Applications,* Oct. 1997.
[36] G. Lopez, "The Design and Implementation of Kaleidoscope, a Constraint Imperative Programming Language," PhD thesis, Univ. of Washington, Apr. 1997.
[37] T. May and M. Woods, "Alpha-Particle-Induced Soft Errors in Dynamic Memories," *IEEE Trans. Electron Devices,* vol. 26, no. 1, 2-9 Jan. 1979.
[38] A. Mishra, J.P. Ros, A. Singhal, G. Weiss, D. Litman, P.F. Patel-Schneider, D. Dvorak, and J. Crawford, "R++: Using Rules in Object-Oriented Designs," *Proc. 11th Ann. Conf. Object-Oriented Programming Systems, Languages and Applications,* July 1996.
[39] S. Mourad and D. Andrews, "On the Reliability of the IBM MVS/XA Operating System," *IEEE Trans. Software Eng.,* Sept. 1987.
[40] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kcman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," Technical Report UCB//CSD-02-1175, Computer Science, Univ. of California Berkeley, Mar. 2002.
[41] J.S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing Under Unix," *Proc. Usenix Winter Technical Conf.,* pp. 213-223, Jan. 1995.
[42] "The Unified Modeling Language," Rational Inc., http://www.rational.com/uml, 2006.
[43] M. Rinard, "Probabilistic Accuracy Bounds for Fault-Tolerant Computations that Discard Tasks," *Proc. 20th ACM Int'l Conf. Supercomputing,* 2006.
[44] M. Rosenblum and J.K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *Proc. 13th ACM Symp. Operating Systems Principles,* Oct. 1991.
[45] P. Shirvani, N.R. Saxena, and E.J. McCluskey, "Software-Implemented EDAC Protection against SEUs," *IEEE Trans. Reliability,* vol. 49, no. 3, pp. 273-284, Sept. 2000.
[46] D. Taylor and J. Black, "Principles of Data Structure Error Correction," *IEEE Trans. Computers,* vol. 31, no. 7, pp. 602-608, July 1982.
[47] D. Taylor, D. Morgan, and J. Black, "Redundancy in Data Structures: Improving Software Fault Tolerance," *IEEE Trans. Software Eng.,* vol. 6, no. 6, pp. 585-594, Nov. 1980.
[48] D. Taylor, D. Morgan, and J. Black, "Redundancy in Data Structures: Some Theoretical Results," *IEEE Trans. Software Eng.,* vol. 6, no. 6, pp. 595-602, Nov. 1980.
[49] D. Taylor, D. Morgan, and J. Black, "A Compendium of Robust Data Structures," *Proc. 11th Int'l Symp. Fault Tolerant Computing,* June 1981.

[50] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro,* Mar./Apr. 2002.

[51] S.D. Urban and L.M. Delcambre, "Constraint Analysis: A Design Process for Specifying Operations on Objects," *IEEE Trans. Knowledge and Data Eng.,* vol. 2, no. 4, Dec. 1990.

[52] E.J. Weyuker, "Using the Consequence of Failures for Testing and Reliability Assessment," *Proc. Third ACM SIGSOFT Symp. Foundations of Software Eng.,* 1995.

[53] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard, "Field Constraint Analysis," *Proce. Int'l Conf. Verification, Model Checking, and Abstract Interpratation,* 2006.

[54] S.S. Yau and R.C. Cheung, "Design of Self-Checking Software," *Proc. Int'l Conf. Reliable Software,* pp. 450-455, 1975.

[55] J.W. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Comm. ACM,* vol. 17, no. 9, pp. 530-531, 1974.

[56] Y. Zhang, D. Wong, and W. Zheng, "User-Level Checkpoint and Recovery for LAM/MPI," *ACM SIGOPS Operating Systems Rev.,* vol. 39, no. 3, pp. 72-81, 2005.

**Brian Demsky** received the PhD degree in 2006 from the Massachusetts Institute of Technology (MIT) while with the Program Analysis and Compilation Group at the MIT Computer Science and Artificial Intelligence Laboratory. He is an assistant professor in the Department of Electrical Engineering and Computer Science at the University of California at Irvine. His current research interests include software reliability, software engineering, compilation, software debugging, and program understanding.

**Martin C. Rinard** is a professor in the Massachusetts Institute of Technology (MIT) Department of Electrical Engineering and Computer Science and a member of the MIT Computer Science and Artificial Intelligence Laboratory. His research interests have included parallel and distributed computing, programming languages, program analysis, program verification, and software engineering. Much of his current research focuses on techniques that enable software systems to execute successfully in spite of the presence of errors. Results in this area include acceptability-oriented computing (a framework for ensuring that software systems satisfy basic acceptability properties), failure-oblivious computing (a technique for enabling programs to execute successfully through otherwise fatal memory addressing errors), and a technique for providing probabilistic bounds on the accuracy of program outputs in the presence of failures.