Static Specification Analysis for Termination of Specification-Based Data Structure Repair

Brian Demsky
Laboratory for Computer Science
Massachusetts Institute of Technology

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology

***Abstract*— We have developed a system that accepts a specification of key data structure consistency constraints, then dynamically detects and repairs violations of these constraints. It is possible to write specifications that are not satisfiable or that for other reasons may cause the repair process to not terminate. We present a static specification analysis that determines whether the repair process will terminate for a given specification.**

I. Introduction

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

Several very successful systems use data structure inconsistency detection and repair techniques to improve reliability in the face of software errors. For example, the Lucent 5ESS switch and IBM MVS operating systems both use hand-coded audit and repair procedures to recover from these errors [16], [19]. The reported results indicate an order of magnitude increase in the reliability of these systems [12]. Similar repair procedures exist for persistent data structures such as file systems and application files.

We have developed a new specification-based approach to the data structure consistency problem [9]. Instead of developing ad-hoc, hand-coded procedures, the developer provides a specification of key data structure consistency properties. Our tool processes this specification to automatically generate code that detects, then repairs, any inconsistent data structures. Our overall goal is to place repair techniques on a more solid formal foundation, to decrease the effort required to obtain a repair system for a given program, and to increase the reliability and predictability of the repair process.

A complication is that the developer may provide an unsatisfiable specification or a specification whose repair algorithm may not terminate. This paper presents a static analysis that, when provided with an arbitrary specification, determines if the corresponding repair algorithm will always terminate. In addition to ensuring that the repair process will not loop forever, this termination also guarantees that the specification is satisfiable.

A. Detection and Repair

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

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

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

B. Static Termination Analysis

Note that the repair actions for one constraint may cause another constraint to become violated. If there is a cycle in which one constraint may be repaired only to become violated by another future repair, the repair process may not terminate. We therefore statically analyze the set of constraints to verify the absence of cyclic repair chains that might result in infinite repair loops. If a specification contains cyclic repair chains, the tool attempts to prune conjunctions to eliminate the cycles.

C. Experience and Contributions

We have used our tool to repair inconsistencies in four applications: an air-traffic control system, a simplified Linux file system, an interactive game, and Microsoft Word files. All of our specifications were statically verified to generate repair procedures that terminated. Furthermore, in all of our benchmark executions, the repair procedure terminates and successfully repairs the data structures.

This paper makes the following contributions:

- **Termination Analysis:** It presents a static specification analysis that determines if the repair process will always terminate for a given specification.
- **Proof:** It presents a proof that acyclicity of the graph constructed by the static specification analysis implies termination of the corresponding repair algorithm.
- **Termination Assurance:** It presents an algorithm that, when possible, eliminates repair choices that may lead to infinite repair loops. This algorithm may convert a repair algorithm that may not terminate to a more restrictive repair algorithm that always terminates.
- **Experimental Results:** The paper describes our experiences writing specifications and the results of using our inconsistency detection and repair tool for several applications.

The remainder of the paper is structured as follows. Section II presents an example that we use to illustrate our approach. Section III presents the specification language used to express the consistency constraints. Section IV presents the inconsistency detection and repair algorithms. Section V presents the termination analysis and gives a correctness proof. Section VI presents our experience using automatic data structure repair in several benchmark applications. Section VII discusses related work. Section VIII presents future work; we conclude in Section IX.

II. Example

We next present an example that illustrates our approach. The data structure in the example implements a list that associates object identifiers with attributes. Figure 1 presents the structure definitions for this data structure. Each node has a field `objectId` that stores the object identifier, a field `numAttributes` that indicates the number of attributes stored in the node, an array `firstAttributes` that stores the first `N` attributes for the object (note that `N` has the value 5), a boolean field `expanded` that signals the presence of an expansion array that may contain an additional M attributes, and a field `restAttributes` that optionally contains a reference to the expansion array.

In our example, `N` and `M` are constants, but we support more advanced declarations in which such quantities could be stored in data structure fields. Figure 2 presents an (inconsistent) instance of the data structure. The `numAttributes` field of the second node in the list has the value 12, which indicates the presence of more attributes than will fit in a node, but the `expanded` flag is set to false indicating that no expansion array is present. Furthermore, the `restAttributes` pointer is set to null as no expansion array is present. Figure 3 presents the data structure after repair (the repair algorithm has changed the `numAttributes` field to 5).

```
#define N 5
#define M 10
struct node {
  node *next;
  int objectId;
  int numAttributes;
  int firstAttributes[N];
  bool expanded;
  rest *restAttributes;
}
struct rest {
  int restAttributes[M];
}
node *attributeList;
```

Fig. 1. Structure Definitions

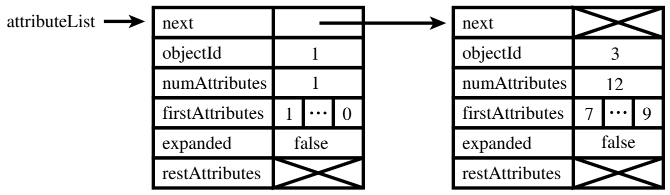

Fig. 2. Inconsistent Data Structure

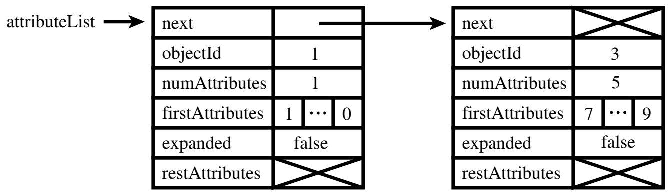

Fig. 3. Repaired Data Structure

We focus on the following consistency constraints:

1) **Flag Consistency:** The `expanded` flag should only be set if the expansion array is present.
2) **Number of Attributes Consistency:** If `numAttributes` is greater than `N`, the `expanded` flag should be set and the expansion array should be present. Furthermore, the number of attributes should not exceed the total attribute capacity of the node (`N+M` attributes).
3) **List Structural Consistency:** No list node should have more than one incoming `next` reference.

To support the expression of these kinds of constraints at an appropriate level of abstraction, we allow the developer to specify a translation from the concrete data structure representation into an abstract model based on relations between objects. The developer can then use this model to state some of the desired consistency constraints.

A. Model Construction

Figure 4 presents the object and relation declarations for our example. There are two sets of objects: `nodes` (with a subset `first`) and `rests`. The `next` relation models chains of list `nodes`. The `expanded` relation models the

```
set nodes of node: subset first;
set rests of rest;
expanded: nodes -> boolean;
restAttributes: nodes -> rest;
numAttributes: nodes -> int;
next: nodes -> nodes;
```

Fig. 4. Object and Relation Declarations

```
[], true => attributeList in first;
[for n in nodes], !n.next=NULL =>
        <n, n.next> in next;
[for n in nodes], !n.next=NULL =>
        n.next in nodes;
[for n in nodes], !n.restAttributes=NULL =>
        n.restAttributes in rests;
[for n in nodes], !n.restAttributes=NULL =>
        <n,n.restAttributes> in restAttributes;
[for n in nodes], true =>
        <n,n.numAttributes> in numAttributes;
[for n in nodes], true =>
        <n,n.expanded> in expanded;
```

Fig. 5. Model Definition Declarations and Rules

`expanded` flag status of the object. The `restAttributes` relation maps `nodes` to the corresponding `rests` objects. The `numAttributes` relation maps `nodes` to the number of attributes they contain.[1]

Figure 5 presents the model definition rules. Each rule consists of a quantifier that identifies the scope of the rule, a guard whose predicate must be true for the rule to apply, and an inclusion constraint that specifies either an object that must be in a given set or a tuple that must be in a given relation. Our tool processes these rules to produce an algorithm that, starting from the `attributeList` variable, traces out the `next` relation and computes the sets of `nodes` and `rests` objects.

B. Consistency Constraints and Repair Algorithm

The developer uses the model to state the data structure consistency requirements: we call such constraints *internal* constraints. Figure 6 presents the internal constraints for our example. The first constraint states that if the expanded flag is set for a `node`, then the `node` must have an expansion array. The second constraint states that the expanded flag must be set for each `node` with more than `N` attributes. The third constraint states that each `node` has at most `N+M` total attributes. The fourth constraint states that each `node` has at most one reference from the `next` field.[2] The fifth constraint ensures that the `numAttributes` and `expanded` relations

[1]In this example the model corresponds quite closely to the concrete data structure, however this is not always the case. In general, we have found the model translation useful for two purposes. First, it supports the clean expression of important relationships hidden in many low-level, heavily encoded data structures. Second, it supports the use of synthesized abstract relations that are not directly present in the data structure but that facilitate the expression of important consistency properties.

[2]We use the notation `next.n` to refer to the image of `n` under the inverse of the `next` relation, i.e., the set of all objects `o` such that $\langle o, n \rangle \in$ `next`.

```
[for n in nodes], n.expanded=false or
        size(n.restAttributes)=1;
[for n in nodes], n.numAttributes<=N or
        n.expanded=true;
[for n in nodes], n.numAttributes<=N+M;
[for n in nodes], size(next.n)<=1;
[for n in nodes], size(n.numAttributes)=1 and
        size(n.expanded)=1;
[], size(first)=1;
```

Fig. 6. Internal Consistency Constraints

are functions. The final constraint ensures that there is at least one `node` in the list.

The repair algorithm repeatedly traverses the model to find a constraint and a set of variable bindings that falsify the constraint. It then executes repair actions that update the model so that the constraint is satisfied for that variable binding. In our example, the repair algorithm detects that the second `node` in the list in Figure 2 violates the second constraint — the `numAttributes` field is greater than `N`, and the `expanded` flag is false. Assume that the repair algorithm repairs this violation by setting the `expanded` flag in this node to `true`. But this repair causes the node to violate the first constraint. If the repair action for this newly introduced inconsistency sets the `expanded` flag back to `false`, the repair algorithm is trapped in an infinite repair loop.

In some cases, restricting the repair choices may ensure termination. In our example, repairing violations of the second constraint by setting the `numAttributes` field to be less than or equal to `N` will ensure that the repair always terminates.

C. Reasoning About Termination

Our algorithm uses an *interference graph* to reason about the termination of the repair process. The nodes in this graph correspond to the conjunctions in the disjunctive normal form of the constraints; there is a directed edge between two conjunctions if the repair action for an atomic formula of the first conjunction may falsify the second conjunction, or if the repair action may increase the scope of one of the quantifiers of the second conjunction. If there are no cycles in this graph, then the repair process will eventually terminate.

Figure 7 presents the interference graph for our example.[3] The possibility of an infinite repair loop shows up as a cycle between the `n.expanded=true` and `n.expanded=false` nodes.

Our algorithm attempts to eliminate such cycles by removing nodes (and their incident edges) from the graph, subject to the constraint that it must leave at least one node per constraint in the graph. In our example, the algorithm eliminates the cycle by removing the `n.expanded=true` node. Figure 8 presents the resulting acyclic graph.

[3]In this figure, the conjunctions from a constraint whose disjunctive normal form has multiple conjunctions appear together within the same dotted box. The figure shows that `(n.numAttributes<=N or n.expanded=true)` is the disjunctive normal form of the second constraint.

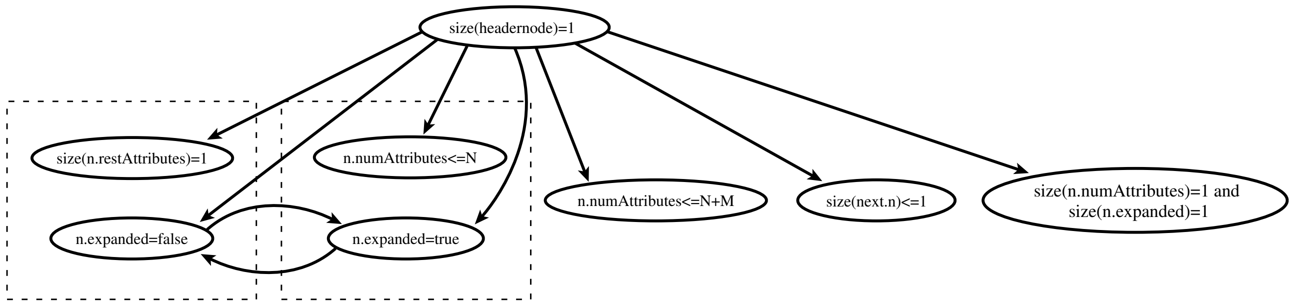

Fig. 7. Interference Graph With Cycle

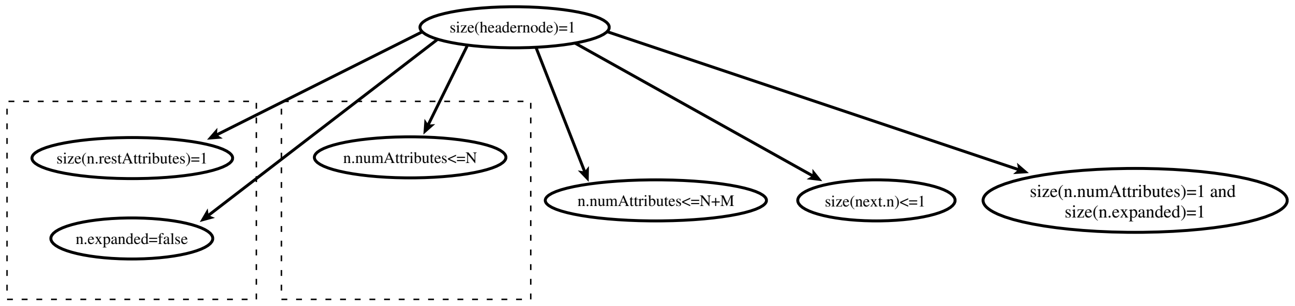

Fig. 8. Interference Graph After Cycle Elimination

We translate the node removals into repair algorithm restrictions by simply preventing the repair algorithm from executing the repair actions designed to satisfy the conjunction corresponding to the node. If a constraint containing a conjunction corresponding to a removed node is violated, the restricted repair algorithm simply satisfies a different conjunction from that constraint.

D. External Constraints

External constraints may reference both the model and the concrete data structures. Figure 9 presents the external constraints in our example. These constraints capture the requirements that the sets and relations in the model place on the values in the concrete data structures. Repairs that enforce these constraints translate the model repairs into concrete data structure repairs by overwriting any inconsistent values in the data structure. In our example, the external constraints cause the `numAttributes` field to be set to 5.

```
[for n in first] true => attributeList=n;
[for <n,nl> in next], true => n.next=nl;
[for <n,e> in expanded], true =>
              n.expanded=e;
[for <n,r> in numAttributes], true =>
              n.numAttributes=r;
[for <n,e> in restAttributes], true =>
              n.restAttributes=e;
```

Fig. 9. External Consistency Constraints

III. Specification Language

Our specification language consists of several sublanguages: a structure definition language, a model definition language, and the languages for the internal and external constraints.

A. Model Definition Language

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

The model definition rules define a translation from the concrete data structures into an abstract model. Each rule has a quantifier that identifies the scope of the rule, a guard whose predicate must be true for the rule to apply, and an inclusion constraint that specifies either an object that must be in a given set or a tuple that must be in a given relation. Figure 10 presents the grammar for the model definition rules.

B. 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`

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

Fig. 10. Model Definition Rule Language

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 invalid pointers. We next describe how we extend the model construction algorithm to deal with invalid pointers.

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

We next extend the model construction software to check that each `struct` accessed via a pointer is valid before it inserts the `struct` into a set or a relation. All valid `struct`s reside completely in allocated memory. In addition, if two valid `struct`s overlap, one must be completely contained within the other and the declarations of both `structs` must agree on the format of the overlapping memory. This approach ensures that only valid `struct`s appear in the model.

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

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

C. Internal Constraints

Figure 11 presents the grammar for the internal constraint language. Each constraint consists of a sequence of quantifiers $Q_1, ..., Q_n$ followed by body B. The body B uses logical connectives (and, or, not) to combine propositions P.

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

Fig. 11. Internal Constraint Language

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

Fig. 12. External Constraint Language

D. External Constraint Language

Figure 12 presents the grammar for the external constraint language which is used to translate model repairs to the concrete data structures. Each constraint has a quantifier that identifies the scope of the rule, a guard G that must be true for the constraint to apply, and a condition C that specifies either a program variable, a field in a structure, or an array element must have a given value.

IV. Error Detection and Repair

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

A. Error Detection in the Internal Phase

The algorithm detects violations of the internal constraints by evaluating the constraints in the context of the model. This evaluation iterates over all values of the quantified variables, evaluating the body of the constraint for each possible combination of the values. If the body evaluates to false, the algorithm has detected a violation and has computed a set of bindings for the quantified variables that make the constraint false.

B. Error Repair in the Internal Phase

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

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

1) Size Propositions: Size propositions are of the form `size`$(SE) = 1$, `!size`$(SE) = 1$, `size`$(SE) >= 1$, or `size`$(SE) <= 1$ where SE can be one of the sets in the model or a relation expression of the form $R.V$ or $V.R$. It is straightforward to generalize size propositions to involve arbitrary constant sizes.

If SE is a set in the model, the repair action simply adds or removes elements to satisfy the constraint. The action ensures that these changes respect any partition and subset constraints between sets in the model. Note that this basic approach also works for negated size propositions.

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

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

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

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

3) Inclusion Propositions: Inclusion propositions are of the form V `in` SE where SE is a set in the model or a relation expression. The repair actions simply add or remove (in the case of negation) V to the set or the appropriate pair to the relation. They then perform corresponding updates to other sets to restore the partition and subset constraints in the model definition.

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

C. 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.

1) Repair Costs: The first mechanism is based on a repair cost associated with each atomic formula. At each step, the repair algorithm must choose one of several violated constraints to repair. Each constraint can be represented as a disjunction of conjunctions; repairing any of these conjunctions will ensure that the constraint is satisfied. The repair of each conjunction, in turn, requires the execution of a repair action for each of its violated atomic formulas. The repair algorithm sums the costs for each of the repair actions, then chooses the conjunction with the least repair cost.

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

We have isolated the choice of which violated constraint to repair inside a separate procedure in our implementation.

It is straightforward to allow the developer to provide us with a partial implementation of this procedure — each time there is a choice to be made, our system would invoke the developer's implementation, which would return a subset of the choices that it found acceptable. Our system would then use the repair costs to choose the least costly alternative from within that subset. In principle, this mechanism gives the developer complete control over the choice should he or she choose to exert this control. To obtain even more control, the developer could specify a hand-coded repair procedure to invoke when the constraint is violated. When the hand-coded repair terminates, the system would verify that the constraint is satisfied, then (once again under developer control) optionally invoke its own standard repair algorithm if the hand-coded repair failed to satisfy the constraint.

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

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

3) Critical Constraints: In some cases, the developer may wish to identify *critical constraints*, or constraints that are so crucial to the continued successful execution of the program that if they are violated, the best strategy is to simply terminate or suspend the execution and await external intervention. We allow the developer to flag such constraints in the specification. If the consistency checker finds that a critical constraint is violated, it suspends the program.

V. Termination Analysis

The acyclicity checking algorithm first converts the body of each constraint into disjunctive normal form. It then constructs an *interference graph*. There is one node in the graph for each conjunction in the disjunctive normal form of each constraint. These nodes are arranged into clusters, with exactly one cluster for each constraint. A cluster contains all of the nodes for the conjunctions making up the constraint.[4] The graph contains the following edges:

- **Quantifier Scope:** There is an edge from a conjunction to a second conjunction if repairing one of the atomic formulas in the first conjunction may increase the scope of a quantifier of the second conjunction. This can happen in two ways: the repair may add an object to the quantifier's set or the repair may change the value of one of the bounds that defines the integer range of the quantifier.
- **Interference:** There is an edge from one conjunction to another conjunction if applying an action to repair one of the atomic formulas in the first conjunction may falsify one of the atomic formulas in the second conjunction. The foundation of this construction is a procedure that determines if one atomic formula may *interfere* with another, i.e., if repairing the first atomic formula may falsify the second. The interference checking algorithm first checks if the two atomic formulas involve disjoint parts of the model; if so, they do not interfere. If the two atomic formulas may involve the same state, the algorithm reasons about the specific repair action and the atomic formula. If the repair action is guaranteed to leave the model in a state that satisfies the second atomic formula, there is no interference. [5]

[4]The clusters for the example (Figure 7) are surrounded by dashed boxes.

A. Rules For Computing Interference

In this section, we formally present the interference graph. The internal constraint specification consists of a set of constraints, $C_1...C_n$. Each of these constraints may be represented in disjunctive normal form as a disjunction of conjunctions, $C_i = Q_1, Q_2, ..., Q_m \bigvee_j d_j$ with conjunctions $d_j = \bigwedge_k \beta_{kj}$, quantifiers Q_i, and each atomic formula β_{kj} is either a proposition P or its negation.

The algorithm constructs an interference graph G from these constraints. We show that acyclicity of this graph implies termination of the repair procedure. We represent the graph as a set of pairs of nodes $G \subseteq N \times N$, where N is the set of nodes. Each node corresponds to one unique conjunction: given a conjunction d_j, $\mathcal{M}(d_j)$ is the corresponding node.

We define the function $\mathcal{QS}(V) = S$ to map the variable V to the set S that the variable V quantifies over. If the variable quantifies over an integer range, the map returns a special token that is contained in no other set. We define the functions $\mathcal{D}$ and $\mathcal{R}$ to map relations to their corresponding domain and range sets, respectively.

1) Set Expansion Interference: When the repair algorithm repairs a constraint violation, the algorithm may add new elements to some of the sets in the model. As a result of this addition, some of the quantifiers may bind to new elements, increasing the number of constraints involving the element that must be satisfied.

a) Subset and Partition Implications: Before discussing the procedure for determining whether a given atomic formula extends the scope of a quantifier, we need to define a variety of functions that abstract various portions of our system.

[5]This is true if the first atomic formula implies the second. It may also be true even in some cases when the second atomic formula implies the first. For example, the two constraints $\texttt{size}(S) >= 1$ and $\texttt{size}(S) = 1$ do not interfere — the repair action for $\texttt{size}(S) >= 1$ makes $\texttt{size}(S) = 1$.

We define the function $\mathcal{IN}$ to capture the transitive closure of the subsetting relations. $\mathcal{IN}(S_1, S_2)$ is `true` if and only if the subsetting relations require that $\forall e.e \in S_1 \Rightarrow e \in S_2$.

The repair algorithm must keep set additions consistent with the partition and subsetting relations between sets. This requirement can lead to unexpected interference between an atomic formula and a quantifier. For example, if the algorithm inserts an element into the set S, where S is partitioned into S_1 and S_2, it must insert the element into either S_1 or S_2. The exact set the element is inserted in may depend on the guidance given by the developer. We model any set additions due to this algorithm by using the function $\mathcal{IS}$. $\mathcal{IS}(S)$ is the set that an element is inserted into when the specification requires that the element be added to set S.

A similar situation exists if the algorithm attempts to remove an element from S_1. To satisfy the partition requirement, in addition to removing the element from S_1, the algorithm must either remove the element from S or insert the element into S_2. The choice the algorithm makes between these options depends on the guidance given by the developer. We model any set additions due to this algorithm using the function $\mathcal{RS}$. $\mathcal{RS}(S)$ is the set that an element is inserted into when the specification requires that the element be removed from set S. If the element is not inserted into any set, $\mathcal{RS}(S)$ returns a special token which is not contained in any other set.

Satisfying a size proposition for a set may require the addition of a new element to the set. The repair algorithm has to have a source for these new elements. The repair algorithm may find an element in another set or the algorithm may use an allocator to create a new element. We define the function $\mathcal{SS}$ to map a set involved in a size proposition to the possible set that serves as a source for new elements. If the source is a memory allocator, the map returns a special token that is contained in no other set.

b) Determining Interference: We use an interferes function $\mathcal{IF}$ to tell the analysis whether coercing the atomic formula β may increase the size of the set S appearing in the quantifier `for` V `in` S. We construct the function $\mathcal{IF}(\beta, \texttt{for}\ V\ \texttt{in}\ S)$ to have the property that if $\mathcal{IF}(\beta, \texttt{for}\ V\ \texttt{in}\ S) = \texttt{false}$ then coercing the atomic formula β to be true does not add any elements to the set S.

In Figure 13, we define the interferes function $\mathcal{IF}$ for all combinations of the atomic formula β and quantifiers over sets (we omit similar rules for quantifiers of the form `for` $V = E_1..E_2$ for brevity).

If repairing the conjunction d adds an element to a set quantified over in a constraint C, the interference graph must contain an edge from the node corresponding to the conjunction d to each of the nodes corresponding to the conjunctions in the constraint C. Formally, for a conjunction $d = \bigwedge_k \beta_k$ and a constraint $C = Q_1, Q_2, ..., Q_m \bigvee_j d'_j$, if there exist x, y such that $\mathcal{IF}(\beta_x, Q_y) = \texttt{true}$ then $\forall j, \langle \mathcal{M}(d), \mathcal{M}(d'_j)\rangle \in G$.

2) Proposition Interference Rules: In the process of repairing a violation of a constraint, the repair algorithm may violate another constraint. As a result of this action, the repair algorithm would have to make an additional repair to satisfy the second constraint.

$\mathcal{IF}(V_1\ \texttt{in}\ S_1, \texttt{for}\ V_2\ \texttt{in}\ S_2)$	is `true` iff $\mathcal{IN}(\mathcal{IS}(S_1), S_2) \wedge \neg\mathcal{IN}(\mathcal{QS}(V_1), S_2)$
$\mathcal{IF}(!V_1\ \texttt{in}\ S_1, \texttt{for}\ V_2\ \texttt{in}\ S_2)$	is `true` iff $\mathcal{IN}(\mathcal{RS}(S_1), S_2)$
$\mathcal{IF}(V_1\ \texttt{in}\ V_2.R, \texttt{for}\ V_3\ \texttt{in}\ S)$	is `true` iff $(\mathcal{IN}(\mathcal{IS}(\mathcal{R}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{QS}(V_1), S)) \vee (\mathcal{IN}(\mathcal{IS}(\mathcal{D}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{QS}(V_2), S))$
$\mathcal{IF}(V_1\ \texttt{in}\ R.V_2, \texttt{for}\ V_3\ \texttt{in}\ S)$	is `true` iff $(\mathcal{IN}(\mathcal{IS}(\mathcal{R}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{QS}(V_2), S)) \vee (\mathcal{IN}(\mathcal{IS}(\mathcal{D}(R), S) \wedge \neg\mathcal{IN}(\mathcal{QS}(V_1), S))$
$\mathcal{IF}(\texttt{size}(S_1) = 1, \texttt{for}\ V\ \texttt{in}\ S_2)$	is `true` iff $(\mathcal{IN}(\mathcal{IS}(S_1), S_2) \wedge \neg\mathcal{IN}(\mathcal{SS}(S_1), S_2)) \vee \mathcal{IN}(\mathcal{RS}(S_1), S_2)$
$\mathcal{IF}(\texttt{size}(S_1) <= 1, \texttt{for}\ V\ \texttt{in}\ S_2)$	is `true` iff $\mathcal{IN}(\mathcal{RS}(S_1), S_2)$
$\mathcal{IF}(\texttt{size}(S_1) >= 1, \texttt{for}\ V\ \texttt{in}\ S_2)$	is `true` iff $\mathcal{IN}(\mathcal{IS}(S_1), S_2) \wedge \neg\mathcal{IN}(\mathcal{SS}(S_1), S_2)$
$\mathcal{IF}(\texttt{size}(V_1.R) = 1, \texttt{for}\ V_2\ \texttt{in}\ S)$	is `true` iff $(\mathcal{IN}(\mathcal{IS}(\mathcal{D}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{QS}(V_1), S)) \vee (\mathcal{IN}(\mathcal{IS}(\mathcal{R}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{SS}(\mathcal{R}(R)), S))$
$\mathcal{IF}(\texttt{size}(V_1.R) >= 1, \texttt{for}\ V_2\ \texttt{in}\ S)$	is `true` iff $(\mathcal{IN}(\mathcal{IS}(\mathcal{D}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{QS}(V_1), S)) \vee (\mathcal{IN}(\mathcal{IS}(\mathcal{R}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{SS}(\mathcal{R}(R)), S))$
$\mathcal{IF}(\texttt{size}(R.V_1) = 1, \texttt{for}\ V_2\ \texttt{in}\ S))$	is `true` iff $(\mathcal{IN}(\mathcal{IS}(\mathcal{R}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{QS}(V_1), S)) \vee (\mathcal{IN}(\mathcal{IS}(\mathcal{D}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{SS}(\mathcal{D}(R)), S))$
$\mathcal{IF}(\texttt{size}(R.V_1) >= 1, \texttt{for}\ V_2\ \texttt{in}\ S)$	is `true` iff $(\mathcal{IN}(\mathcal{IS}(\mathcal{R}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{QS}(V_1), S)) \vee (\mathcal{IN}(\mathcal{IS}(\mathcal{D}(R)), S) \wedge \neg\mathcal{IN}(\mathcal{SS}(\mathcal{D}(R)), S))$

Fig. 13. Computing Interference Between Atomic Formulas and Quantifiers

We use the interferes function $\mathcal{PI}$ to tell the analysis whether coercing one atomic formula may falsify another atomic formula. We construct the interferes function to have the property that $\mathcal{PI}(\beta_1, \beta_2) = \texttt{false}$ implies that coercing the atomic formula β_1 to be true will not falsify the atomic formula β_2. The algorithm uses a case analysis to calculate the interferes function $\mathcal{PI}$.

If repairing one conjunction may falsify another, the graph must contain a directed edge between the two conjunctions. The algorithm adds an edge from the conjunction containing β to the conjunction containing β' if $\mathcal{PI}(\beta, \beta') = \texttt{true}$. Formally, for $d = \bigwedge_k \beta_k$ and $d' = \bigwedge_{k'} \beta'_{k'}$ if there exist x, y such that $\mathcal{PI}(\beta_x, \beta'_y) = \texttt{true}$ then $\langle \mathcal{M}(d), \mathcal{M}(d')\rangle \in G$.

B. Acyclicity Implies Termination

We next show that the acyclicity of the interference graph implies that the repair process always terminates. Some possible complications in the repair process are the possibility that a given conjunction may need to be repaired multiple times (for different quantifier bindings) and the possibility that a given repair may increase the number of quantifier bindings of other constraints.

Theorem 1. *Repairs for a system of constraints $C_1, ..., C_n$ terminate if the corresponding interference graph G is acyclic.*
Proof Sketch. (Structural induction).
(Base Case:) The base case, an acyclic graph of size 0, terminates because there are no violated conjunctions.
(Induction Step:) We assume that repairs terminate on all acyclic graphs of size k or less. We must show that all repairs terminate for an acyclic graph of size $k+1$.

Since the graph is acyclic, it must contain a node n with no incoming edges. Furthermore, all nodes in the same cluster (nodes corresponding to conjunctions from the same constraint) have no incoming edges arising from a possible quantifier scope expansion. Otherwise the node n would have a similar incoming edge as it shares the same quantifiers with the other nodes in the cluster. Because there are no incoming edges to node n, the algorithm repairs each quantifier binding for n at most once — once the node is satisfied for a given quantifier binding, no other repair will falsify it. Therefore, the conjunction represented by node n may only be repaired a number of times equal to the number of quantifier bindings for the constraint that the conjunction appears in.

By the induction hypothesis, repairs on acyclic graphs of size k terminate. So after each repair of node n the algorithm either eventually repairs all violations of conjunctions corresponding to the other k nodes (leaving only violations of the conjunction corresponding to node n to possibly repair) or it repairs a violation of the node n before finishing the repairs on the other nodes. Since the conjunction represented by node n may only be repaired a number of times equal to the number of quantifier bindings for the constraint the conjunction appears in, the repair must eventually terminate.

C. Checking Algorithm

The termination checking algorithm first checks to see if the interference graph is acyclic. If it is not acyclic, it searches for a set of nodes to remove from this graph in an attempt to make the graph acyclic. Note that it must leave at least one node in the graph for each constraint. Once a node is removed from the graph, it is marked as forbidden to ensure that the repair algorithm never chooses to repair an inconsistency by satisfying the conjunction corresponding to that node. In general, it may not be possible to produce an acyclic interference graph, in which case the termination checking algorithm rejects the specification.

D. Specification Limitations

In some cases, the termination analysis may reject a specification for which repairs would terminate. For example, the repair algorithm might not realize that the repair it uses to satisfy one algebraic constraint automatically satisfies another algebraic constraint involving the same value. In practice, we don't expect this to significantly limit the range of desired specifications. It is also possible, however, to increase the power of the atomic formula interference rules mentioned in Section V-A.2 to recognize more algebraic properties.

E. Repair Limitations

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

The internal constraint repair algorithm will fail only because of resource limitations — i.e., if it is unable to find an element or tuple to add to a set or relation, either because it is unable to allocate a new `struct` or because there are no more distinct elements in the set that it is using as a source of new elements. The external constraint repair algorithm will fail only if the external constraints specify different values for the same data structure value — in this case, the algorithm will produce a data structure with only one of the values.[6]

The static cyclicity check described in Section V rules out many potential failure modes, in particular, it eliminates the possibility of unsatisfiable specifications.

VI. Experience

We next discuss our experience using our implemented repair tool to detect and repair inconsistencies in data structures from several applications: an air-traffic control system, a Linux file system, an interactive game, and Microsoft Office files. We have a complete implementation of the data structure repair tool. The implementation consists of 13,000 lines of C++ code and is available (with specifications for the benchmarks) at http://www.cag.lcs.mit.edu/∼bdemsky/repair.[7]

A. Methodology

For each application, we identified important consistency constraints and developed a specification that captured these constraints. We ran the static analysis on these specifications to verify that the repairs for these specifications would terminate. We also developed a fault insertion strategy designed to simulate the effect of potential inconsistencies.[8] We applied the fault insertion strategy to the data structures in the applications, then both verified termination of the repair algorithm and compared the results of running a chosen workload with and without inconsistency detection and repair. We ran the applications on an IBM ThinkPad X23 with a 866 Mhz Pentium III processor and 384 MB of RAM. For the Linux file

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

[7]As CTAS is deployed air traffic control software, we are not permitted to distribute it. As such, it is excluded from the archive.

[8]Fault insertion was originally developed in the context of software testing to help evaluate the coverage of testing processes [26]. It has also been used by other researchers for the purposes of evaluating standard failure recovery techniques such as duplication, checkpointing, and fast reboot [2]. The rationale behind fault insertion is that faults, while serious when they do occur, occur infrequently enough to seriously complicate the experimental investigation of failure recovery techniques. Fault insertion makes it practical to evaluate proposed recovery techniques on a range of faults.

system and the interactive game application, we used RedHat Linux 7.2. For the Microsoft Office file application, we used Microsoft Office XP running on the Microsoft Windows XP operating system.

B. CTAS

The Center-TRACON Automation System (CTAS) is a set of air-traffic control tools developed at the NASA Ames research center [1], [24]. The system is designed to help air traffic controllers visualize and manage the complex air traffic flows at centers surrounding large metropolitan areas.[9] In addition to graphically displaying the location of the aircraft within the center, CTAS also uses sophisticated algorithms to predict aircraft trajectories and schedule aircraft landings. The goal is to automate much of the aircraft traffic management, reducing traffic delays and increasing safety. The current source code consists of over 1 million lines of C and C++ code. Versions of this source code are deployed at various centers (Dallas/Ft. Worth, Los Angeles, Denver, Miami, Minneapolis/St. Paul, Atlanta, and Oakland) and are in daily use at these centers.

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

The CTAS software maintains data structures that store aircraft data. Our experiments focus on the flight plan objects, which store the flight plans for the aircraft currently within the center. These flight plan objects contain both an origin and destination airport identifier. The software uses these identifiers as indices into an array of airport data structures. Flight plans are transmitted to CTAS as a long character string. The structure of this string is somewhat complicated, and parsing the flight plan string to build the corresponding flight plan data structure is a challenging activity.

Our fault insertion methodology attempts to mimic errors in the flight plan processing routine that produce illegal values for the airport identifier fields in the flight plan data structures. Our specification captures the constraint that the flight plan indices must be within the bounds of the airport data array. The specification itself consists of 100 lines, of which 83 lines contain structure definitions. The primary obstacle to developing this specification was reverse engineering the source (which consists of over 1 million lines of C and C++ code) to develop an understanding of the flight plan data structures. Once we understood the data structures, developing the specification was straightforward.

[9]A center is a geographical region surrounding a metropolitan area. There are 20 centers in the 48 contiguous states; each center may contain multiple major airports along with several smaller regional airports. Because these airports share overlapping airspaces, the air traffic flows must be coordinated for all of the aircraft within the center, regardless of their origin or destination.

The static analysis verified that the repair algorithm generated by this specification terminates. This is because the action of repairing one constraint in this application does not violate any other constraints. Our experience running the application confirms the results of the static analysis.

We used a recorded midday radar feed from the Dallas-Ft. Worth center as a workload. We identified consistency points within the application, then configured the system to catch addressing exceptions, perform the consistency checks and repair in the fault handler, then restart from the last consistency point. Each consistency check and repair takes approximately 3 milliseconds, which is an acceptable repair time in that it imposes no performance degradation that is visible in the graphical user interface that displays the aircraft information.

Without repair, CTAS fails because of an addressing exception. With repair, it continues to execute in a largely acceptable state. Specifically, the effect of the repair is to potentially change the origin or destination airport of the aircraft with the faulty flight plan processing. Even with this change, continued operation is clearly a better alternative than failing. First, one of the primary purposes of the system (visualizing aircraft flow) is unaffected by the repair, and continued execution enables the system to provide this functionality to the controller even in the presence of flight plan processing errors. Second, only the origin or destination airport of the plane whose flight plan triggered the error is affected. All other aircraft (during the recorded feed, the system is processing flight plans for several hundred aircraft) are processed with no errors at all, enabling the system to deliver useful trajectory prediction and scheduling functionality for those aircraft. And finally, once the aircraft in question leaves the center, its data structures are deallocated from the system, which is then back to a completely correct state. One improvement that would further improve the utility of the repaired system is a way to visually identify aircraft with repaired flight plan information. We are currently exploring ways to leverage existing GUI functionality to make this happen.

C. A Linux File System

Our Linux file system application implements a simplified version of the Linux ext2 file system [22]. The file system, like other Unix file systems, contains bitmaps that identify free and used disk blocks [11]. The file system uses these disk blocks to support fast disk block and inode allocation operations.

Our consistency constraints are that the inode and block bitmap blocks, the directory block, and the inode table blocks exist; and that these blocks are consistent with each other and a variety of other constraints. The specification contains 122 lines, of which 53 lines contain structure definitions. Because the structure of such file systems is widely documented in the literature, it was relatively easy for us to develop the specification.

The static termination analysis verified that repairs for the Linux file system specification would always terminate. The constraints in this example are not independent as the action of repairing one constraint may result in the violation of another constraint. However, the dependency graph for this example is a tree. For any given action to repair a constraint, only a finite number of additional repair actions are required to repair any additional constraints that the original repair may have violated. We found that the restrictions that the termination analysis places on the constraints did not interfere with writing the specification.

In all of our tested cases, the algorithm is able to repair the file system and the workload correctly runs to completion. Without repair, files end up sharing inodes and disk blocks and the file contents are incorrect. For a file system with 1024 8KB blocks, our repair tool takes 1.5 seconds to construct the file system model, check the consistency of the model, and repair the file system.

D. Freeciv

Freeciv is an interactive, multi-player game available at www.freeciv.org. The Freeciv server maintains a map of the game world. Each tile in this map has a terrain value chosen from a set of legal terrain values. Additionally, cities may be placed on the tiles. Our consistency constraints are that tiles have valid terrain values, a given city has exactly one location, cities are not in the ocean, and that the location of a city on the map is consistent with the location the city has recorded internally.

The specification consists of 218 lines, of which 173 lines contain structure definitions. The primary obstacle to developing this specification was reverse engineering the Freeciv source (which consists of 73,000 lines of C code) to develop an understanding of the data structures.

Although the constraints for the Freeciv example are not independent (the repair action for one constraint may violate another), the constraint dependency graph is a tree. As a result, no infinite repair chains are possible, and the static termination analysis verifies that repairs for the Freeciv specification always terminate. We found that the restrictions that the termination analysis places on the constraints did not interfere with developing the specification.

In all of the test games, our repair tool terminated and was able to successfully repair the introduced inconsistencies, enabling the game to execute to completion (although the game played out differently because of changed terrain values). For a map of 4,000 tiles, our repair tool took 6.7 seconds to construct the model, check its consistency, and repair the game map.

E. Microsoft Office File Format

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

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

The static termination analysis verified that repairs for the Word document's specification would terminate. This is because the action of repairing one constraint in this application does not violate any other constraints. We found that the restrictions that the termination analysis places on the constraints did not interfere with writing a specification. In our test cases, the repair tool terminated for each damaged Word file and was able to successfully repair the file. Without repairs, Word refused to read most of the damaged files. For a 150KB file, our repair tool takes 8.4 seconds to construct the model, check the consistency of the model, and repair the file.

VII. Related Work

Software reliability has been an important area for many years. Most research has focused on preventing or eliminating software errors, with the approaches ranging from enhanced software testing and validation to full program verification. Software error detection has become an especially active area in recent years [7], [8], [15], [6].

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

A. Manual Detection and Repair Systems

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

The Lucent 5ESS telephone switch and IBM MVS operating systems are two examples of critical systems that use inconsistency detection and repair to recover from software failures [16], [19]. The software in both of these systems contains a set of manually coded procedures that periodically inspect their data structures to find and repair inconsistencies. The reported results indicate an order of magnitude increase in the reliability of the system [12]. Researchers have also developed a domain-specific language for specifying these procedures for the 5ESS system [14]. The goal is to enhance the reliability and reduce the development time of the inconsistency detection and repair software. The 5ESS system has also served as the platform for PRL5, a declarative constraint specification language [18], and its compiler, which generates code to automatically check the consistency of a relational database used to store some it its information [13]. The compiler can also generate, for each operation, the weakest precondition required to ensure that the operation preserves the consistency constraints. Although the generated code does not perform any repairs, the consistency checking alone is valuable enough to justify its presence.

These successful, widely used systems illustrate the utility of performing inconsistency detection and repair. We see our use of declarative specifications coupled with automatically generated detection and repair code as representing a significant advance over current practice, which relies on the manual development of the detection and repair code. Our approach enables the developer to focus on the important data structure constraints rather than on the operational details of developing algorithms that detect and correct violations of these constraints. We believe our specification-oriented approach will make it much easier to develop reliable inconsistency detection and repair software. It also places the field on a firmer foundation, since it is based on a set of properties that the repair algorithm is designed to deliver rather than on a set of hand-coded repair routines whose effect may be more difficult to determine.

B. Integrity Maintenance in Databases

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

One approach is to provide a system that assists the developer in creating a set of production rules that maintain the integrity of a database [5]. Each production rule consists of a triggering component and a repair action to execute when the rule is triggered. The system automatically generates the triggering components of the production rules, using a triggering graph to check if repairs will terminate. The system relies on the developer to provide the actual repair actions; if the developer incorrectly specifies a repair action, the system may fail to maintain the integrity of the database.

This approach has been extended to enable the system to automatically generate both the triggering components and the repair actions [4]; the resulting system can automatically generate repairs that insert or remove tuples to or from a relation. The specification language can express similar properties as our internal constraint language, but the termination analysis is less precise. For some constraints the system may generate production rules that fail to terminate. For example, the system cannot automatically generate terminating repairs for a system of constraints that require a relation to be a function, then further constrain this function. Because of differences in the repair algorithms, our system is able to enforce these kinds of constraints.

Researchers have also developed a database repair system that enforces Horn clause constraints and schema constraints (which can constrain a relation to be a function) [25]. The system includes an interactive tool, which can help developers understand the consequences of repairing constraint violations. Our system supports a broader class of constraints — logical formulas instead of Horn clauses. It also supports constraints which relate the value of a field to an expression involving the size of a set or the size of an image of an object under a relation. Finally, it uses partition information to improve the precision of the termination analysis, enabling the verification of termination for a wider class of constraint systems.

It is also possible to apply constraint enforcement to structured documents [20]. This system accepts a set of consistency properties expressed in first-order logic, generates a set of repair actions for each constraint, and then interactively queries the user to select a specific repair action for violated constraints. Because the system performs no termination analysis, it is possible for infinite repair cycles to occur.

C. Self-Stabilizing Algorithms

Researchers in the area of self-stabilizing algorithms have developed distributed algorithms that eventually converge to a stable state in spite of perturbations [10]. Our research goal differs in that 1) we aim to provide a general-purpose, specification-based inconsistency detection and repair technology for arbitrary data structures (as opposed to designing individual algorithms with desirable constraints), and 2) we are willing to accept potentially degraded behavior as the price of obtaining this generality. In some cases, however, our data structure repair algorithm may make the global program behave in a self-stabilizing way. In particular, if the effect of the repair is eventually flushed out of the system (as in the CTAS benchmark), the data structures eventually converge back to a state that has no trace of the error or the repair.

D. Traditional Error Recovery

Error recovery has been an important topic ever since the inception of computer science as a field. One standard approach avoids transient errors by simply rebooting the system; this is perhaps the most widely practiced form of error recovery. Checkpointing enables a system to roll back to a previous state when it fails. Transactions support consistent atomic operations by discarding partial updates if the transaction fails before committing [12]. Database systems use a combination of logging and replay to avoid the state loss normally associated with rolling back to a previous checkpoint. In effect, the log serves as a redundant, very simple data structure that can be used to rebuild the more sophisticated internal database data structures whenever they become inconsistent. There has recently been renewed interest in applying many of these classical techniques in new computational environments such as Internet services [21]. One of the techniques that arises in this context, recursive restartability, composes large systems out of many smaller modules that are individually rebootable [3]. The goal is to build systems in which faults can be isolated at the module level by rebooting.

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

Our approach can enable systems to recover even from persistent errors such as file system corruption. Unlike approaches based on checkpointing and replay, it may preserve much of the volatile state and avoids the need for logging and replay. It can also keep a system going without the need to take it out of service while it is rebooting. Finally, our approach differs in that we do not attempt to recover to a state that a (hypothetical) correct program would produce. Instead, our goal is to recover to a state consistent enough to permit the continued operation of the program within its design envelope. In many cases, the system will, over the course of time, flush the effects of errors out of its data structures and return to a completely correct state.

E. Specification Languages

The core of our specification language is the internal constraint language. The basic concepts in this language (objects and relations) are the same as in object modeling languages such as UML [23] and Alloy [17], and the constraint language itself has many of the same concepts and constructs as the constraint languages for these object modeling languages, which are specifically designed, in part, to be easy for developers to use. In addition to these ease of use considerations, the relative simplicity of the basic object modeling approach facilitates the automatic repair process. Because all structural properties are expressed in terms of cardinality constraints involving sets of objects and relations, it is possible to repair violations of these constraints by simply removing or inserting objects or pairs of objects into sets or relations.

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

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

VIII. Future Work

Our current approach separates the repair process into two activities: application of the internal constraints to repair the model followed by application of the external constraints to translate the model repairs into actions that repair the concrete data structures. It is currently the responsibility of the developer to ensure that the external constraints correctly translate the model repairs — if the external constraints are incorrect, the repaired data structures may be arbitrarily inconsistent even though the model satisfies all of its internal constraints. In particular, running the model construction algorithm on the repaired data structures may deliver a new and different model that violates the internal consistency constraints.

Note also that the external constraints are largely redundant — the model definition rules by themselves should, in principle, contain all the information necessary to establish the connection between the data structures and the model. We are therefore developing a technique that uses goal-directed reasoning on the model definition rules to eliminate the external consistency constraints altogether. This technique will guarantee the connection between the model and the data structures, eliminating potential anomalies caused by the use of external consistency constraints.

Separating the internal and external consistency constraints in our current formulation substantially simplifies the termination analysis — this separation enables the termination analysis algorithm to reason only about the effect of actions that operate on the model. The application of the external consistency constraints takes place after the model repair, and the algorithm that applies these constraints always terminates.

Using goal-directed reasoning to replace the external consistency constraints would force the termination analysis to consider the connection between the data structures and the model. In addition to reasoning about the effect of actions that update the model, it would also have to reason about the interaction between the data structure updates and the model updates. We see this additional complexity as (one of) the inevitable technical difficulties associated with eliminating the external consistency constraints.

IX. Conclusion

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

Furthermore, we believe that such systems should provide some degree of confidence that they will successfully repair damaged data structures. Our algorithm provides actions that can successfully repair any violated constraint; our analysis guarantees that a repair algorithm that uses these actions will always terminate. These facts may increase the confidence that developers have in the ability of our algorithm to use their specifications to successfully repair their data structures.

We believe that our research provides a principled mechanism for introducing repair into software systems, enabling a more reliable and robust concept of system behavior.

X. Acknowledgments

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

References

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

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

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

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

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

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

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

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

[9] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference of Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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