Automatic Detection and Repair of Errors in Data Structures

Brian Demski, Martin Rinard

April 28, 2005
1 Introduction
   • Scope
   • Approach

2 Example
   • Model Construction
   • Consistency Constraints

3 Specification Language, Check & Repair
   • Structure Definition Language
   • Model Definition Language
   • Constraints
   • Error Detection and Repair

4 Experience
Motivation

The problem

- Programs manipulate data structures
- Software error or anomaly causes inconsistency
- Assumptions under which software was developed no longer hold
  - software behaves in unpredictable manner

The solution proposed

- Do *not* increase the reliability of the code
- Specify key data structure consistency constraints
- Dynamically detect and repair data structures violating the constraints
Motivation

The problem
- Programs manipulate data structures
- Software error or anomaly causes inconsistency
- Assumptions under which software was developed no longer hold
  - software behaves in unpredictable manner

The solution proposed
- Do *not* increase the reliability of the code
- Specify key data structure consistency constraints
- Dynamically detect and repair data structures violating the constraints
Goal

- Do not necessarily restore the data structure to previous consistent state the program was into
- Deliver repaired data structures satisfying the consistency assumptions of the program
  - The program is able to continue to operate successfully

Intended Scope

- Prioritize continued execution even after concrete evidence of error
- Clearly might not be acceptable for all computations
  - *safety-critical systems* like air traffic control can benefit
Goal

- Do not necessarily restore the data structure to previous consistent state the program was into
- Deliver repaired data structures satisfying the consistency assumptions of the program
  - The program is able to continue to operate successfully

Intended Scope

- Prioritize continued execution even after concrete evidence of error
- Clearly might not be acceptable for all computations
  - safety-critical systems like air traffic control can benefit
Basic Approach

Data structure views
- Concrete view – in memory bits
- Abstract view – relations between abstract objects
  - specification of high level constraints
  - reasoning to repair inconsistencies

Specification
- Set of model definition rules
- Set of consistency constraints
Automatic tool

- Generate algorithm that builds the model,
- Inspect the model and data structures to find constraint violations
- Repair data structures

Repair algorithm

- Inconsistency detection
- Converts each violated constraint to DNF (disjunctive normal form)
- Apply repair actions – may cause other constraint to be violated
Automatic tool

- Generate algorithm that builds the model,
- Inspect the model and data structures to find constraint violations
- Repair data structures

Repair algorithm

- Inconsistency detection
- Converts each violated constraint to DNF (disjunctive normal form)
- Apply repair actions – may cause other constraint to be violated
Invoking Check & Repair

- Data structures are updated, may be inconsistent temporarily
- Programmer marks program points where he/she expects data structures to be consistent
- Augment programs to perform check & repair in signal handlers
- Persistent data structures – use a stand alone separate mechanism
An example, FAT

**Figure:** Inconsistent File System

**Figure:** Repaired File System
**FAT Constraints**

- **Chain Disjointness:** Each block should be in at most one chain

- **Free Block Consistency:** No chain should contain a block marked as free

**Abstract Constraints**

- Developer specifies a translation from concrete data structure representation to abstract model

- Express the constraints in terms of the abstract model
**FAT Constraints**

- **Chain Disjointness:** Each block should be in at most one chain
- **Free Block Consistency:** No chain should contain a block marked as free

---

**Abstract Constraints**

- Developer specifies a translation from concrete data structure representation to abstract model
- Express the constraints in terms of the abstract model
**Object and Relation Declarations**

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

**Figure:** Graphical Representation of Object and Relation Declarations
struct Entry {
    byte name[Length];
    byte valid;
    int size;
    int first;
}
struct Block { data byte[BlockSize]; }
struct Disk {
    Entry table[NUMEntries];
    int FAT[NUMBlocks];
    Block block[NUMBlocks];
}
Disk disk;

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

**Figure:** Model Definition Declarations and Rules

**Rule structure**

- quantifier identifying the scope of the rule
- guard that has to be true for the rule to apply
- inclusion constraint – used to build the sets & relations
Internal Consistency Constraints

- *Internal* constraints are stated using the model exclusively
- Do not use the concrete data structures

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

---

Inconsistent Model

Repair Model
External Consistency Constraints

- May reference both model and concrete data structures
- Captures requirements the sets and relations place on the values in the concrete data structures
- Used to translate model repairs back into concrete data structure

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

Sublanguages

- Structure definition language
- Model definition language
- Language for constraints
  - internal constraints
  - external constraints
Declaring the layout of data in memory

\[
\text{structdefn} \quad := \quad \text{struct structurename} \\
\quad (\text{subtypes structurename}) \{\text{fielddefn}^*\}
\]

\[
\text{fielddefn} \quad := \quad \text{type field; | reserved type; |}
\quad \text{type field}[E]; |
\quad \text{reserved type}[E];
\]

\[
\text{type} \quad := \quad \text{boolean | byte | short | int | structurename}
\quad \text{structurename} \ast
\]

\[
E \quad := \quad V \mid \text{number | string | E.field |}
\quad \text{E.field}[E] \mid E - E \mid E + E \mid E/E \mid E \ast E
\]

**Figure:** Structure Definition Language
Model Definition Language

- Define a translation from concrete data structures into an abstract model
- Set declaration set $S$ of $T$ : partition $S_1, \ldots, S_n$
  - partition keyword can be replaced by subsets
- Relation declaration relation $R$: $S_1 \rightarrow S_2$

Given a model containing the rules, set of concrete data structures $h$, naming environment $l$, the model is the least fixed point of the functional:

$$\lambda m. (R[C_1] h \ l) \ldots (R[C_1] h \ l \ m)$$
Model Definition Language

\[
C := Q, C \mid G \Rightarrow I \\
Q := \text{for } V \text{ in } S \mid \text{for } \langle V, V \rangle \text{ in } R \mid \\
\text{for } V = E \ldots E \\
G := G \text{ and } G \mid G \text{ or } G \mid \neg G \mid E = E \mid E < E \mid \text{true} \mid \\
(G) \mid E \text{ in } S \mid \langle E, E \rangle \text{ in } R \\
I := E \text{ in } S \mid \langle E, E \rangle \text{ in } R \\
E := V \mid \text{number} \mid \text{string} \mid E.\text{field} \mid \\
E.\text{field}[E] \mid E - E \mid E + E \mid E/E \mid E \ast E
\]

**Figure:** Model Definition Language
Internal Constraints

- Each constraint is a sequence of quantifiers $Q_1, Q_2, \ldots, Q_n$ followed by body $B$
- $B$ uses logical connectors (i.e. and, or, not) to combine basic propositions $P$
- Constraint $C$ is evaluated in the context of a model $m$ and a naming environment $l$

Complication

For undefined values cannot yield true or false $\Rightarrow$ extend to 3-value logic by introducing maybe
Internal Constraints

- Each constraint is a sequence of quantifiers $Q_1, Q_2, \ldots, Q_n$ followed by body $B$
- $B$ uses logical connectors (i.e. and, or, not) to combine basic propositions $P$
- Constraint $C$ is evaluated in the context of a model $m$ and a naming environment $l$

Complication

For undefined values cannot yield true or false $\Rightarrow$ extend to 3-value logic by introducing maybe
Internal Constraints

\[
C ::= Q, C | B \\
Q ::= \text{for } V \text{ in } S | \text{for } V = E .. E \\
B ::= B \text{ and } B | B \text{ or } B | !B | (B) | \\
    V E = E | V E < E | V E <= E | V E > E | \\
    V E >= E | V \text{ in } SE | \text{size}(SE) = C | \\
    \text{size}(SE) >= C | \text{size}(SE) <= C \\
V E ::= V.R \\
E ::= V | \text{number} | \text{string} | E + E | E - E | E/E | \\
    E * E | E.R | \text{size}(SE) | (E) \\
SE ::= S | V.R | R.V
\]

**Figure:** Internal Constraints Language
External Constraints

Each constraint has

- a quantifier identifying the scope of the rule
- a guard $G$ that must be true for the constraint to apply
- a condition $C$ specifying the value of either
  - a program variable
  - a field in a structure
  - an array element

Constraint $R$ evaluated in the context of naming environment $I$, model $m$ and set of concrete data structures $h$
External Constraints

Figure: External Constraints Language

\[
R := Q, R | G \Rightarrow C \\
Q := \text{for } V \text{ in } S | \text{for } \langle V, V \rangle \text{ in } R | \text{for } V = E .. E \\
G := G \text{ and } G | G \text{ or } G | G | E = E | E < E | \text{true} \\
C := HE.field = E | HE.field[E] = E | V = E \\
HE := V | HE.field | HE.field[E] \\
E := V | \text{number} | \text{string} | E.R | E - E | E + E | E \times E | E/E | \text{size}(SE) | \text{element } E \text{ of } SE \\
SE := S | V.R | R.V
\]
Error Detection and Repair

Detection
Detect violations by evaluating constraints (internal & external) in the context of the model
- Iterates over all values of the quantified variables, evaluating body

Repair
Updates the model and the concrete data structures $\Rightarrow$ all internal & external constraints satisfied
- Repair actions coerce propositions to be true
Error Detection and Repair

Detection
Detect violations by evaluating constraints (internal & external) in the context of the model
- Iterates over all values of the quantified variables, evaluating body

Repair
Updates the model and the concrete data structures \(\Rightarrow\) all internal & external constraints satisfied
- Repair actions coerce propositions to be true
## Error Detection & Repair in Internal Phase

**Repair Algorithm**

- **Input:** a body and variable bindings that falsify the body
- **Output:** change the model to make the body true
  - Converts body to DNF
  - Each basic proposition has an associated repair action
  - Choose one conjunction, apply repair to all basic propositions
Repair Actions

Size propositions

- \( \text{size}(S) = C, \neg \text{size}(S) = C, \text{size}(S) > C \ldots \)
- \( C \) integer constant, \( S \) set or relation expr (\( R.v \) or \( v.R \))
- \( S \) is set \( \Rightarrow \) add or remove items to the set
  - make sure the partition and subset inclusion are preserved
- \( S \) is a relation expr \( \Rightarrow \) add or remove tuples
  - may modify domains

Where do items come from?

- structs – memory allocation primitives; superset
- primitive types – synthesize new values
Repair Actions

Size propositions

- \( \text{size}(S) = C, \neg \text{size}(S) = C, \text{size}(S) > C \ldots \)
- \( C \) integer constant, \( S \) set or relation expr (\( R.v \) or \( v.R \))
- \( S \) is set \( \Rightarrow \) add or remove items to the set
  - make sure the partition and subset inclusion are preserved
- \( S \) is a relation expr \( \Rightarrow \) add or remove tuples
  - may modify domains

Where do items come from?

- structs – memory allocation primitives; supersets
- primitive types – synthesize new values
# Repair Actions

## Inequality Propositions
- $V.R = E$, $!V.R = E$, $V.R < E$, $V.R <= E$, $V.R > E$, $V.R >= E$
- Evaluate $E$, update $V.R$

## Inclusion Propositions
- $V \text{ in } SE$, where $SE$ is a set in the model or a relation expression
- Add or remove value referenced by $V$, obeying partition and subset requirements
**Repair Actions**

**Inequality Propositions**
- $V.R = E$, $!V.R = E$, $V.R < E$, $V.R \leq E$, $V.R > E$, $V.R \geq E$
- Evaluate $E$, update $V.R$

**Inclusion Propositions**
- $V \text{ in } SE$, where $SE$ is a set in the model or a relation expression
- Add or remove value referenced by $V$, obeying partition and subset requirements
Choosing the Conjunction to Repair

- When faced with a choice – use a cost heuristic
  - additive cost function for the repair actions of each proposition in a conjunction
- Designed to minimize the number of changes
- Tuned to discourage removal of objects
  - preserve as much information about original data structures as possible
Termination

Constraint dependence graph

- One node for every constraint and one node for every conjunction in DNF
- Edges:
  - Constraint to Conjunctions
  - Interference
  - Quantifier Scope

acyclic graph ? will terminate : prune conjunctions & and try again
Error Detection & Repair in External Phase

Repair

- Detection algorithm $\Rightarrow$ variable bindings that falsify constraint
- Assign data structure value to be same as model value
Methodology

Implementation
Interpreter to construct model, check for consistency violations and repair

Test subjects
- CTAS – air-traffic control tools
- Simplified version of ext2
- Freeciv – multiplayer game
- Microsoft Office File Format
## Performance

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of model definition rule applications</th>
<th>Total size of sets (objects)</th>
<th>Total size of relations (tuples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTAS</td>
<td>20</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>File system</td>
<td>11720</td>
<td>3128</td>
<td>1954</td>
</tr>
<tr>
<td>Freeciv</td>
<td>63072</td>
<td>7537</td>
<td>15990</td>
</tr>
<tr>
<td>Word</td>
<td>139740</td>
<td>64</td>
<td>17</td>
</tr>
</tbody>
</table>
## Performance

<table>
<thead>
<tr>
<th>Application</th>
<th>Time to construct model (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTAS</td>
<td>4.2</td>
</tr>
<tr>
<td>File system</td>
<td>1,188.9</td>
</tr>
<tr>
<td>Freeciv</td>
<td>5,609.1</td>
</tr>
<tr>
<td>Word</td>
<td>7,189.5</td>
</tr>
</tbody>
</table>

**Table:** Time to construct model
Performance

<table>
<thead>
<tr>
<th>Application</th>
<th>Internal constraint evaluations</th>
<th>Time to check internal constraints (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTAS</td>
<td>4</td>
<td>0.09</td>
</tr>
<tr>
<td>File system</td>
<td>2384</td>
<td>16.6</td>
</tr>
<tr>
<td>Freeciv</td>
<td>16004</td>
<td>175.3</td>
</tr>
<tr>
<td>Word</td>
<td>28</td>
<td>0.2</td>
</tr>
</tbody>
</table>

**Table:** Number of checks and time to check and repair internal constraints
## Performance

<table>
<thead>
<tr>
<th>Application</th>
<th>External constraint evaluations</th>
<th>Time to enforce external constraints (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTAS</td>
<td>4</td>
<td>0.2</td>
</tr>
<tr>
<td>File system</td>
<td>3164</td>
<td>59.5</td>
</tr>
<tr>
<td>Freeciv</td>
<td>12001</td>
<td>171.4</td>
</tr>
<tr>
<td>Word</td>
<td>39</td>
<td>1.2</td>
</tr>
</tbody>
</table>

**Table:** Number of checks and time to enforce external constraints
Fin