Efficient Specification-Assisted Error Localization

Brian Demsky Cristian Cadar Daniel Roy Martin Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
{ bdemsky, cristic, droy, rinard }@mit.edu

ABSTRACT

We present a new error localization tool, Archie, that accepts a specification of key data structure consistency constraints, then generates an algorithm that checks if the data structures satisfy the constraints. We also present a set of specification analyses and optimizations that (for our benchmark software system) significantly improve the performance of the generated checking algorithm, enabling Archie to efficiently support interactive debugging.

We evaluate Archie's effectiveness by observing the actions of two developer populations (one using Archie, the other using standard error localization techniques) as they attempted to localize and correct three data structure corruption errors in a benchmark software system. With Archie, the developers were able to localize each error in less than 10 minutes and correct each error in (usually much) less than 20 minutes. Without Archie, the developers were, with one exception, unable to locate each error after more than an hour of effort.

1. INTRODUCTION

Error localization is a key prerequisite for eliminating programming errors in software systems and, in many cases, the primary obstacle to correcting the error — the fix is often obvious once the developer locates the code responsible for the error.

The primary issue in error localization is minimizing the time between the error and its manifestation as observably incorrect behavior. The greater this time, the longer the program executes in an incorrect state and the harder it can become to trace the manifestation back to the original error. This issue can become especially problematic for data structure corruption errors — these errors often propagate from the original corrupted data structure to manifest themselves in distant code that manipulates other derived data structures, obscuring the original source of the error.

This paper presents a new error localization tool, Archie[1], describes the optimizations required to make Archie efficient enough for practical use, and discusses the results of a case study we performed to evaluate its effectiveness in helping developers to localize and correct data structure corruption errors. Our results indicate that, after optimization, Archie executes efficiently enough for interactive use on our benchmark software system and that it can dramatically improve the ability of developers to localize and correct injected data structure corruption errors in this system.

1.1 Consistency Checking

Consistency checking is currently used as a technique for debugging [17]. Developers sometimes hand-code consistency checks in the same programming language as the rest of the system. The complication is that developers must code data structure traversals and implement any auxiliary data structures required to check the desired properties. Developing this code can be especially difficult because the developer cannot assume that the data structures satisfy any property at all — the whole point of the checker is to detect data structures that may arbitrarily violate their invariants. For example, straightforward hand-coded tree traversals may fail to terminate on trees that contain cycles.

Hand-coded consistency checkers are also vulnerable to anomalies such as incomplete property coverage, unwarranted assumptions about the input data structures, and increased development overhead. Our experience indicates that hand-coded consistency checkers are substantially larger and more difficult to develop than an equivalent consistency specification.

1.2 Specification-Based Approach

Archie accepts a specification of key data structure consistency properties (including sophisticated properties characteristic of complex linked data structures), then periodically monitors the data structures to detect and flag violations of these properties. The developer (potentially assisted by an automated tool) places calls to Archie into the software system. If the system contains a data structure corruption error, Archie localizes the error to the region of the execution between the first call that detects an inconsistency and the immediately preceding call (which found the data structures to be consistent).

Each Archie specification contains a set of model definition rules and a set of consistency properties. Archie (conceptually) interprets these rules to build an abstract model of the concrete data structures, then examines the model to find any violations of the consistency properties. The conceptual separation of the specification into the model construction rules and consistency constraints simplifies the expression of the consistency constraints and provides important expressibility benefits. Specifically, it enables the specification developer to 1) classify objects into different sets and apply different consistency constraints to objects in different sets, 2) express the consistency constraints at the level of the concepts in the domain rather than at the level of the (potentially heavily encoded) realization of these concepts in the concrete data structures, 3) use inverse relations to express constraints on the objects that may refer (either directly or conceptually) to a given object, 4) construct auxiliary relations that allow the developer to express constraints between objects that are separated by many references in the data structures, and 5) express constraints involving abstract relationships such as object ownership.

1.3 Optimizations

It is clearly desirable to perform the consistency checks as frequently as possible to minimize the size of the region of the execution that may contain the error. The primary obstacle is the check execution overhead. We found that our initial implementation of the consistency checking algorithm as described above was too inefficient for practical use. We therefore implemented the following optimizations:

- **Fixed-Point Elimination:** The Archie compiler analyzes the dependences in the specification to, when possible, replace the fixed-point computation in the model construction phase with a more efficient single-pass algorithm.
- **Relation Elimination:** The compiler analyzes the specification to, when possible, replace the explicit construction of each re-

[1] Archie is named after Archie Goodwin, the assistant to Rex Stout's fictional detective Nero Wolfe. The idea is that, under Wolfe's direction, Archie does all the work required to localize the crime to a specific suspect, then Wolfe uses his superior intelligence to solve the crime.

lation with a computation that efficiently generates, on the demand, the required tuples in the relation.

- **Set Elimination:** The compiler analyzes the specification to, when possible, integrate the consistency checking computation for each set of abstract objects into the data structure traversal that (in the absence of optimization) constructs that set. The success of this optimization enables Archie to eliminate the construction of that set.

Together, these optimizations make Archie run over 800 times faster on our benchmark software system than the original compiled version; the fully optimized instrumented version executes less than 6.2 times slower than the original uninstrumented version. For our benchmark software system, the optimized version of Archie is efficient enough to be used routinely during development with more than acceptable performance for interactive debugging.

1.4 Case Study

To evaluate Archie's effectiveness in supporting error localization and correction, we obtained a benchmark software system, used manual fault injection to create three incorrect versions, then asked six developers to localize and correct the errors. Three developers used Archie; the other three used standard techniques.

With Archie, the developers were able to localize each error within several minutes and correct the error in (usually much) less than twenty minutes. Without Archie, the developers were (with a single exception) unable to localize each error after more than an hour of debugging. The key problem was that continued execution made the errors manifest themselves far (in both code and data) from the original source of the error. Although the developers eventually came to understand what was going wrong, they were unable to trace the manifestation back to its root cause within the allotted time.

To place these results in context, consider that our benchmark system contains significant numbers of assertions designed to catch data structure corruption errors, two of the three errors manifest themselves as assertion violations, but these assertions were still not enough to enable the developers to locate the errors in a timely manner. These results indicate that Archie can provide a substantial improvement over standard error localization techniques.

1.5 Contributions

This paper makes the following contributions:

- **Archie:** It presents the design, implementation, and evaluation of Archie, a new specification-based data structure consistency checking tool for error localization and correction.
- **Optimizations:** It presents a set of optimizations (fixed point elimination, relation elimination, and set elimination) that, together, increase the performance of Archie on our benchmark software system by over a factor of 800, enabling Archie to be used routinely during interactive development with more than acceptable performance.
- **Case Study:** It presents a case study that evaluates the effectiveness of Archie as an error localization and correctness tool. With Archie, developers were able to quickly localize and correct errors in our benchmark software system; without Archie, developers were unable to localize the errors even after they spent significant amounts of time attempting to trace the manifestation of the errors back to their root causes.

2. EXAMPLE

We next present an example (inspired by the FreeCiv program discussed in Section 6) that illustrates how Archie works. The program maintains a grid of tiles that implements the map of a multiple-player game. Each tile has a terrain value (i.e. ocean, river, mountain, grassland, etc) and an optional reference to a city that may be built on that tile. Figure 1 presents the relevant data structure definitions.

```
structure city { int population; }
structure tile { int terrain; city *city; }
tile grid[EDGE * EDGE];
```

Figure 1: Structure Definitions

```
set TILE of tile
set CITY of city
relation CITYMAP: TILE -> CITY
relation TERRAIN: TILE -> int
```

Figure 2: Set and Relation Declarations

```
for x=0 to EDGE*EDGE, true => grid[x] in TILE
for t in TILE, true => <t,t.terrain> in TERRAIN
for t in TILE, !t.city = NULL =>
    <t,t.city> in CITYMAP
for t in TILE, !t.city = NULL => t.city in CITY
```

Figure 3: Model Definition Rules

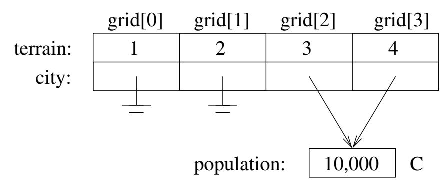

Figure 4: Concrete Data Structure

TILE = $\{grid[0], grid[1], grid[2], grid[3]\}$
TERRAIN = $\{\langle grid[0], 1\rangle, \langle grid[1], 2\rangle, \langle grid[2], 3\rangle, \langle grid[3], 4\rangle\}$
CITY=$\{C\}$
CITYMAP=$\{\langle grid[2], C\rangle, \langle grid[3], C\rangle\}$

Figure 5: Model Constructed for Example

Even a data structure this simple has important consistency constraints; in this section we focus on the following constraints: the terrain field of each tile contains a legal value, each city is referenced by exactly one tile, and no city is placed on an ocean tile.

2.1 Expressing Consistency Properties

To express these constraints, the developer first identifies the sets and relations that conceptually model the concrete data structures. In our example there are two sets, `TILE` and `CITY`, and two relations, `CITYMAP` and `TERRAIN`. Figure 2 presents the declarations of these sets and relations. The `TILE` set contains `tile` structures, and the `CITY` set contains `city` structures. Each relation consists of a set of tuples with objects from two specified sets.

2.1.1 Model Definition Rules

The developer next provides a set of model definition rules that define a translation from the concrete data structures to the sets and relations in the model. Figure 3 presents the model definition rules in our example. Each rule consists of a quantifier that identifies the scope of the rule, a guard that must be true for the rule to apply, and an inclusion constraint that specifies an object (or tuple) that must be in a given set (or relation). Conceptually, Archie uses a least fixed-point algorithm to repeatedly add objects to sets and tuples to relations until the model satisfies all of the constraints. For the data structure in Figure 4, Archie constructs the model in Figure 5.

2.1.2 Consistency Constraints

The developer next uses the sets and relations to state the consistency constraints. Each constraint consists of a sequence of quantifiers that identify the scope of the constraint and a predicate that the constraint must satisfy.

Figure 6 presents the constraints in our example. The first constraint ensures that each tile has a valid terrain, the second ensures that each city has exactly one location (i.e., exactly one tile references each city), and the final constraint ensures that no city is placed on

```
for t in TILE, MIN <= t.TERRAIN and t.TERRAIN <= MAX
for c in CITY,sizeof(CITYMAP.c)=1
for c in CITY,!(CITYMAP.c).TERRAIN = OCEAN
```

Figure 6: Consistency Constraints

an ocean tile. [2] As this example illustrates, the ability to freely use inverses substantially increases the expressive power of the specification language — it enables the expression of properties that navigate backwards through the referencing relationships in the data structures to capture properties that involve both an object and the objects that reference it.

2.2 Instrumentation and Use

Finally, the developer (potentially with the aid of an automated tool) instruments the code to periodically invoke Archie, which examines the data structures and reports any inconsistencies to the developer. When the instrumented program executes, Archie localizes the error to the region of the execution between two subsequent calls to Archie and identifies the violated constraint (which, in turn, identifies the corrupt data structure). Our results (as discussed in Section 6) show that this approach can enable the developer to quickly localize and correct the error that caused the inconsistency. With standard approaches, the program typically continues its execution for some period of time, with the error propagating through the data structures. This combination of continued execution and error propagation makes it difficult to understand and localize the error.

3. SPECIFICATION LANGUAGE

Our specification language consists of several sublanguages: a structure definition language, a model definition language, and a model constraint language.

3.1 Structure Definition Language

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

3.2 Model Definition Language

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

The model definition rules define a translation from the concrete data structures into an abstract model. Each rule has a quantifier that identifies the scope of the rule, a guard whose predicate must be true for the rule to apply, and an inclusion constraint that specifies an object (or tuple) that must be in a given set (or relation). Figure 7 presents the grammar for the model definition language.

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

Figure 7: Model Definition Language

In principle, the presence of negation in the model definition language opens up the possibility of unsatisfiable model definition rules. We address this complication by requiring the set of model definition rules to have no cycles that go through rules with negated inclusion constraints in their guards.

3.3 The Constraint Language

Figure 8 presents the grammar for the model constraint language. Each constraint consists of a sequence of quantifiers $Q_1, ..., Q_n$ followed by body B. The body uses logical connectives (and, or, not) to combine basic propositions P that constrain the sets and relations in the model. Developers use this language to express the key consistency constraints.

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

Figure 8: Model Constraint Language

4. COMPILATION AND OPTIMIZATION

We implemented a compiler that processes Archie specifications to generate C code that implements a basic consistency checking algorithm. This algorithm first uses a work-list-based fixed point algorithm to construct the model, then evaluates the consistency constraints to detect any possible inconsistencies. Unfortunately, this straightforward compilation strategy generates checking algorithms that are too slow for our purposes. We therefore implemented the following optimizations.

4.1 Fixed Point Elimination

This optimization analyzes the model definition rules to replace, when possible, the fixed point computation with a more efficient data structure traversal. The compiler first performs a dependence analysis on the model definition rules to generate a dependence graph. This graph captures the dependences between rules which create sets and relations and the rules which use those sets and relations. Formally, the graph consists of a set of nodes N (one for each rule) and a set of edges E. There is an edge $E = \langle N_1, N_2\rangle$ from N_1 to N_2 if N_2 *uses* a set or relation that N_1 *defines*. A rule *uses* a set S (or a relation R) if the rule has a quantifier of the form `for` V `in` S (or of the form `for` $\langle V_1, V_2\rangle$ `in` R) or if the rule has a guard of the form E `in` S (or $\langle E_1, E_2\rangle$ `in` R). A rule *defines* a set S (or relation R) if it has an inclusion constraint I of the form E `in` S (or $\langle E_1, E_2\rangle$ `in` R).

[2]Note that the notation `CITYMAP.c` denotes the inverse image of `c` under the relation `CITYMAP` (the set of all `t` such that $\langle \mathtt{t}, \mathtt{c}\rangle$ in `CITYMAP`).

The compiler topologically sorts the strongly connected components in the dependence graph. For components that consist of a single rule, the compiler generates efficient code that iterates through all of the rule's possible quantifier bindings, evaluates the guard for each binding, and (if the guard is satisfied) executes the actions that add the appropriate objects to sets or tuples to relations. For components that consists of multiple rules, the compiler generates code that performs a fixed point computation of the sets and relations that the component produces. The generated code executes the computations for the components in the topological sort order. This order ensures that each set and relation is completely constructed before it is used to construct additional sets and relations in other components.

4.2 Relation Elimination

Some of the relations constructed in our model correspond to partial functions. For example, a field f may generate a relation that relates each object o to the value of the field $o.f$. Our compiler discovers relations that implement partial functions and verifies that these relations are used only in the forward direction (i.e., no expression uses the inverse of the relation). The compiler recognizes that a relation R is a partial function if the model definition rules use a single rule of the following form to define R:

$\texttt{for}\ V\ \texttt{in}\ S,\ G\ \Rightarrow\ \langle V, E\rangle\ \texttt{in}\ R.$

The compiler rewrites each expression that uses a partial function by replacing the use with the computation of G and (if G is satisfied) E. The compiler then removes the rule responsible for constructing each such relation.

4.3 Set Elimination

Our final optimization attempts to transform the specification to eliminate set construction and instead perform the checks directly on the data structures in memory. We use two transformations: *model definition rule inlining* and *constraint inlining*. Model definition rule inlining finds a model definition rule of the form $Q^*,\ G_1\ \Rightarrow\ V_1\ \texttt{in}\ S$, a second model definition rule of the form $\texttt{for}\ V_2\ \texttt{in}\ S,\ G_2 \Rightarrow I$, then eliminates the use of the set S in the second rule by transforming it to $Q^*,\ G_1 \wedge G_2[V_1/V_2] \Rightarrow I[V_1/V_2]$. To apply the transformation, the first rule must be the only rule that defines S.

The constraint inlining transformation finds a model definition rule of the form $Q^*,\ G\ \Rightarrow\ V_1\ \texttt{in}\ S$, a consistency constraint of the form $\texttt{for}\ V_2\ \texttt{in}\ S, C$, then eliminates a use of the set S by transforming the consistency constraint to $Q^*,\ G\ \Rightarrow\ C[V_1/V_2]$. To apply the transformation, the model definition rule must be the only rule that defines S. Note that the new constraint has a predicate ($G\ \Rightarrow\ C[V_1/V_2]$) that may involve both concrete values from the data structures in memory and the sets and relations in the model. We have extended the internal representation of our compiler so that it can generate code to check these kinds of hybrid constraints.

Each transformation eliminates a use of the set S. If the transformations eliminate all uses, the compiler removes the set and the rule that produces the set from the specification, eliminating the time and space required to compute and store the set. This optimization can be especially useful when (as is the case for our benchmark system) the compiler is able to eliminate the largest sets or relations.

4.4 Performance Impact

Table 1 presents the execution times of our benchmark software system with the consistency checks at different optimization levels. As these numbers show, the optimizations produce dramatic performance improvements. The final optimized version is more than efficient enough for interactive debugging use.

5. ENVISIONED USAGE STRATEGY

Obtaining developer acceptance of a new tool can be difficult, especially when the tool requires the developers to use a new language

Version	**Time**
No Instrumentation	0.234 sec
Baseline Compiled	20 min
Fixed point elimination	25.60 sec
Relation Elimination	10.66 sec
Set removal	1.45 sec

Table 1: Performance Results

such as our specification language. We expect that several aspects of Archie will facilitate its acceptance within the developer community:

- **Black Box Usage:** The specifications can be developed by a small number of developers who are familiar with the specification language, while the remainder of the developers can simply use Archie as a black box. We anticipate no need for the vast majority of the developers to learn the Archie specification language. There is also no need to change the programming language, coding style, or other development tools.
- **Incremental Adoption:** Archie supports incremental adoption — the developer can start with a specification that captures a small subset of the consistency properties, then incrementally augment the specification to capture more properties. During the specification development process the consistency checker becomes more useful as more properties are added. Calls to Archie can also be incrementally added to the system. The overall result is a smooth integration into the development process with no major dislocations or disruptions.
- **Ease of Development:** Based on our experience developing similar specifications in another project [7], we believe that Archie specifications will prove to be relatively easy to develop once the developer understands the relevant data structures.[3] Because the specifications identify global data structure invariants rather than specific properties of local computations, our experience indicates that the resulting specifications are quite small (the largest are several hundred lines long, with the majority of the lines devoted to structure definitions) in comparison with the size of the software system as a whole.

We do anticipate that the use of Archie may wind up substantially changing the testing, error localization, and error correction activities, but in a positive way — we anticipate that Archie will help developers find errors earlier and provide them with substantially improved error localization. The developers in our case study (see Section 6) had no problem integrating Archie into their debugging strategy and in fact used Archie almost immediately to eliminate tedious activities such as augmenting the code with print statements or using a debugger to insert breakpoints and examine the values of selected variables.

We expect that Archie will effectively support usage strategies in which the initial specifications are developed as part of the software design process before coding begins and usage strategies in which it is integrated into a large existing software system. We also anticipate that, once integrated, the developers will be motivated to keep the specification up to date to reflect changes to the data structures. The division of the specification into model definition rules and consistency constraints facilitates this specification maintenance — if only the representation of the data changes, the developer can simply update the model definition rules to reflect the new representation, leaving the consistency constraints intact.

During development, we expect the program to be instrumented with calls to the Archie consistency checker. We anticipate two kinds

[3] Specifically, we have developed specifications for the FreeCiv interactive game, the CTAS air-traffic control system [1, 23] (this deployed system consists of over 1 million lines of code), a simplified version of the Linux ext2 file system [20], and Microsoft Word files.

of instrumentation: calls placed (potentially with the aid of an automatic call placement tool) at standard locations such as procedure entry and exit points as a routine part of the development process, and calls placed at chosen locations by developers as they attempt to localize a specific error.

6. CASE STUDY

Our case study attempts to answer the most basic question one could ask about Archie's potential effectiveness: Given a specification and a data structure corruption error that causes the data structures to violate this specification, does Archie help developers localize and correct the error? To answer this question, we obtained a benchmark software system and a population of developers, then performed a study in which the developers attempted to localize and correct errors in the system. By comparing the behavior and effectiveness of the developers that used Archie with the developers that did not, we are able to obtain an indication of how well Archie aided the error localization and correction process for this class of errors.

6.1 Developer Population

We recruited six developers with relatively homogeneous backgrounds: all developers had similar educational backgrounds, all represented their home country in international programming competitions while they were in high school, and all are currently students at MIT.

We separated the developers into two populations: the Tool population, which used Archie during the debugging experiments, and the NoTool population, which did not use Archie. To control for debugging ability, we assigned each developer a pre-study calibration task of locating and correcting an error in a heapsort implementation. We ordered the developers by the time required to correct this error; the times varied between 9 and 32 minutes. We then randomly assigned one of the first two, the next two, and the last two developers to the Tool population, with the others assigned to the NoTool population.

6.2 FreeCiv

We chose the FreeCiv interactive game program (available at `http://www.freeciv.org`) as our benchmark software system. The source code consists of roughly 65,000 lines of C in 142 files. It contains four modules: a server module, a client module, an AI module, and a common module. We have made all of the information required to replicate our results available at `http://www.mit.edu/~cristic/Archie`.

6.2.1 Consistency Properties

FreeCiv maintains a map of tiles arranged as a rectangular grid. Each tile contains a terrain value (plains, hills, ocean, desert, etc.) and a reference to a bitmap which maintains additional information (such as pollution levels) about the tile. Each tile may also contain a reference to a city data structure. Our FreeCiv specification consists of 199 lines (of which 180 contain structure definitions). This specification identifies the following five consistency properties: each game must have a single map, each game must have a single grid of tiles, each tile must have a valid terrain value, exactly one tile must point to each city, and no city may be located on an ocean tile.

6.2.2 Incorrect Versions

We used manual fault insertion to create three incorrect versions of FreeCiv. The first version contains an error in the common module. The incorrect procedure is 14 lines long (after error insertion); the error causes the program to assign an invalid terrain value to a tile (causing the data structures to violate the third constraint identified above). The second version contains an error in the server module. The incorrect procedure is 18 lines long and causes two tiles to refer to the same city (causing the data structures to violate the fourth constraint). The third version also contains an error in the server module. The incorrect procedure is 153 lines long; the error causes a city to be placed on an ocean tile (violating the last constraint).

6.2.3 Experimental Setup

We first presented all of the developers with a FreeCiv tutorial, which gave them an overview of the purpose and structure of the program, an overview of Archie, and an overview of the FreeCiv data structures and their consistency properties.

We gave both the Tool and NoTool populations identical instrumented copies of the three incorrect versions of FreeCiv. These copies contain calls to the Archie consistency checker at the beginning and end of each procedure, with the exception of small procedures like structure field getters and setters and I/O procedures that interface with the user or the network. For the NoTool population, these calls immediately return without performing any consistency checking; for the Tool population, each call uses the Archie specification to perform a complete consistency check. Consistent with the expected usage strategy in Section 5, the Tool developers used Archie as a black box — they simply compiled the pre-generated consistency checker into their executables.

The instrumented versions of FreeCiv contain approximately 750 statements that invoke the Archie consistency checker. For the Tool population, each call (whether it detects an inconsistency or not) writes an entry to a log indicating the position in the code from which it was invoked. For this study, we configured FreeCiv to use its autogame mode in which it plays against itself and set the random number generator seed to a fixed value (to ensure repeatability). In this mode, the correct version of the program invokes the checker more than 20,000 times when it executes.

We asked the developers to attempt to locate and eliminate the errors in the three incorrect versions. We requested that they spend at least one hour on each version and allowed them to spend more time if they desired. For the NoTool population, each error manifested itself as either an assertion violation (the first two errors) or a segmentation fault (the last error). For the Tool population, each error manifested itself as an error message from the Archie consistency checker — the consistency checker printed out the violated constraint, the location of the call to the consistency checker, and an explanation of the error provided by the developer of the specification.

All of the developers used a Linux workstation (RedHat 8.0 Linux) with two 2.8 GHz Pentium 4 processors and 2 GBytes of RAM. We provided all of the developers with scripts to compile and run the three versions. The developers were able to use any development or debugging tool available on this platform. The developers were all familiar with this computational environment and comfortable using it. We observed the developers during the experiment and maintained a detailed record of their actions.

6.3 The Tool Population

Table 2 presents the number of minutes required for each member of the Tool population to locate each error; Table 3 presents the total number of minutes required to both locate and correct the error. As these numbers show, the developers were able to locate and correct the errors quite rapidly.

The developers in this population used Archie extensively in their debugging activities. They all started by examining the Archie inconsistency message. If the message came from a call to the Archie consistency checker at the start of a procedure, they examined the Archie log to find the caller of this procedure and (correctly) attributed the error to the caller. If the message came from a call to the Archie consistency checker at the end of a procedure, they (once again correctly) attributed the error to this procedure.

Participant	Error 1	Error 2	Error 3
T1	1	2	1
T2	2	3	2
T3	5	1	5

Table 2: Localization Times (Tool)

Participant	Error 1	Error 2	Error 3
T1	9	7	3
T2	8	6	8
T3	17	7	14

Table 3: Correction Times (Tool)

They then examined the message to determine which constraint was violated, then examined the code of the procedure containing the error to find the code responsible for the inconsistency. For the third error (recall that the procedure containing this error is 153 lines long) the developers inserted additional calls to the Archie consistency checker to further narrow down the source of the inconsistency. Eventually all of the developers found and eliminated the error.

6.4 The NoTool Population

Table 4 presents the number of minutes required for each member of the NoTool population to locate each error; Table 5 presents the total number of minutes required to both locate and correct the error. A dash (-) indicates that the developers were unable to locate or correct the error; a number in parenthesis after the dash indicates the number of minutes spent on the respective task before giving up. As these tables indicate, only one of the developers was able to locate and correct an error. Moreover, this correction was somewhat fortuitous: the developer spent the last 15 minutes of his attempt to locate the second error examining the correct version of the procedure that was modified to contain the third error. When he reexamined this procedure during his attempt to locate the third error, he noticed that the code was different and simply replaced the incorrect version with the correct version that he had examined earlier!

Participant	Error 1	Error 2	Error 3
NT 1	-	-	10
NT 2	-	-	-
NT 3	-	-	-

Table 4: Localization Times (NoTool)

Participant	Error 1	Error 2	Error 3
NT 1	- (95)	- (65)	11
NT 2	- (90)	- (70)	- (60)
NT 3	- (70)	- (60)	- (60)

Table 5: Correction Times (NoTool)

For the first two versions of FreeCiv, the developers in the NoTool population started by examining the code that triggered the assert violation. For the third version, the developers started their examination with the code that triggered the segmentation fault. Once it became clear to them that the code surrounding the assertion or segmentation fault was not responsible for the inconsistency, they attempted to trace the execution backwards to locate the code responsible for the error. During this process, they made extensive use of gdb to set break points and examine the values of the program variables. They also inserted print statements to track the values of different variables and augmented the program with additional assertions to check various consistency properties. Our observations indicate that all of the developers in this group made meaningful progress towards localizing the error. But because of the complexity of the program and the time between the generation of the inconsistency and its manifestation, they were unable to successfully localize the error within the amount of time they were willing to spend.

After several days we asked the developers in the NoTool population to attempt to use Archie to localize and correct the errors. Tables 6 and 7 present the localization and correction times, respectively.[4] As these results show, once the NoTool developers were given access to Archie, they were able to quickly localize and correct the errors.

Participant	Error 1	Error 2	Error 3
NT 1	1	2	-
NT 2	3	2	1
NT 3	3	1	8

Table 6: Localization Times (NoTool with Archie)

Participant	Error 1	Error 2	Error 3
NT 1	2	3	-
NT 2	4	3	6
NT 3	4	3	19

Table 7: Correction Times (NoTool with Archie)

6.5 Discussion

Error localization was the crucial step for debugging the errors in our study and Archie's ability to detect and flag each inconsistency immediately after it was generated was primarily responsible for the divergent experiences of the two populations. Developers in both populations had a clear manifestation of the error and started the debugging process by examining the code that produced this manifestation. For the Tool population, Archie produced a manifestation that quickly directed each developer to the procedure containing the incorrect code. Once directed to this procedure, the developers were able to quickly and effectively locate and correct the error.

	Significant Procedure Calls	Execution Time (%)
Error 1	12689	15%
Error 2	579	1%
Error 3	4142	8.5%

Table 8: Error to Manifestation Distance

Without Archie, the program executed for a substantial period of time before the data structure inconsistency finally manifested itself as an assertion violation or segmentation fault. Table 8 presents numbers that quantify this distance. The first column presents the number of significant procedure calls (this number excludes getter, setter, and I/O procedure calls) between each error and its manifestation as an assertion violation or segmentation fault; the second column presents this distance as a percentage of the running time of the correct version.

Moreover, the inconsistency did not cause incorrect code to fail — it instead caused distant correct code to fail, misleadingly directing the developer to fruitlessly examine correct code instead of incorrect code as the source of the error. Even though the NoTool population was able to obtain a reasonably accurate understanding of each error, their inability to localize the error (even given their understanding) prevented them from correcting it. And once the NoTool population was given access to Archie, they were able to use Archie to quickly and effectively locate and correct the error.

[4]There are no results for developer NT 1 on error 3 because this developer localized and corrected this error in the previous experiment.

6.5.1 Comparison With Assertions

Our results reveal several limitations of assertions as a debugging tool. Like Archie, assertions test basic consistency constraints and, if a constraint is violated, tell the developer which property was violated and where in the execution the violation was detected. It is therefore not clear that Archie should provide any benefit for a program whose assertions successfully detect inconsistencies. But in our study, Archie proved to be substantially more useful to the developers than the assertions, *even though two out of the three data structure inconsistencies manifested themselves as assertion violations*. There are two (related) reasons for this (counterintuitive) result: 1) the assertions in FreeCiv detected the inconsistencies long after their generation, and 2) the assertions did not direct the developers to inconsistencies in the initially corrupted data structures — they instead directed them to inconsistencies in data structures derived from the initially corrupted data structures.

The assertions in FreeCiv, as in many other programs, tend to test easily available values accessed by the surrounding code. The assertions therefore test only partial, local properties of the accessed parts of the data structure, typically properties that the code containing the assertion relies on for its correct execution. In particular, if a computation reads some data structures and produces others, the assertions tend to test the read data structures, not the produced data structures.

It is therefore possible (and even likely) for a program to execute successfully through many assertions after it corrupts its data structures. And when an assertion finally catches the inconsistency, the execution may be very far away from the code responsible for the inconsistency and the inconsistency may have propagated through additional data structures. In our incorrect versions of FreeCiv, for example, one phase of the program produces an inconsistent data structure, but the assertions detect these inconsistencies only after a distant phase attempts to read a data structure derived from the original inconsistent data structure — the intervening phases either do not attempt to access this data structure or fail to check for the violated consistency property.

Because Archie comprehensively checks all of the consistency properties, it makes the developer aware of the inconsistency as soon as it occurs. This immediate notification was crucial to its success in our study, because (unlike the delayed notification characteristic of the existing FreeCiv assertions) it immediately directed the developers to the incorrect code and identified the data structure that it corrupted (and not some other derived data structure).

6.5.2 Efficiency

The basic benefit of Archie is to localize each error to the region of the execution between the failed consistency check and the immediately preceding successful consistency check. It is therefore desirable to perform the consistency checks as frequently as possible so as to better localize the error. The primary obstacle to frequent consistency checking is the overhead of executing the checks.

The optimizations discussed in Section 4 are therefore crucial to the successful use of Archie. Without optimization, the consistency checks increase the FreeCiv execution time from less than a second to twenty minutes. While this kind of time dilation may be acceptable for errors that would otherwise be very difficult to localize, we would prefer to enable developers to use Archie routinely during all of their executions.

Our optimizations enabled us to provide the developers in our study with a checker that can execute frequently while maintaining an interactive debugging environment. We believe that this level of efficiency was crucial to the successful use of Archie in our study and that our optimizations will prove to be at least as important for obtaining an acceptable combination of check frequency and response time for other applications.

6.5.3 Applicability

Our study indicates that consistency checking in general and Archie in particular can help developers locate and eliminate data structure corruption errors that violate the checked consistency property. For this class of errors, Archie provides the developer with information that helps the developer to both localize the bug and understand the violated consistency properties.

We believe that Archie is less likely to be useful for finding errors that do not result in data structure corruption, although it could still be useful for ruling out classes of errors. However, from our experiences we believe that data structure corruption errors are a particularly difficult class to debug, and that Archie should prove useful in practice for this class of errors.

6.5.4 Future Work

This study leaves many interesting questions unanswered. In particular, it provides no indication whether a specification-based approach provides any advantages or suffers from any disadvantages as compared with an approach based on manually developing consistency checkers in the standard implementation language. We anticipate that in either case an expert would develop the specification or consistency checker, most developers would use the consistency checker as a black box, and the development of the consistency checker would require a small fraction of the overall development time. Potential advantages of the specification-based approach include reduced development time, a clearer and more explicit identification of the important consistency properties, and consistency checkers with fewer errors that are amenable to static analysis and targeted optimizations (such as those discussed in Section 4). It remains to be seen if these potential advantages materialize in practice.

A second area of potential inquiry concerns the frequency, relative importance, and consistency violation properties of data structure corruption errors in practice. Our study leaves open questions of whether data structure corruption errors are an important problem in practice and whether developers are able to produce specifications or consistency checkers that catch the kinds of data structure corruptions that occur in practice.

7. RELATED WORK

Error localization and correction has been an important issue ever since people began to develop software. Researchers have developed a host of dynamic and static debugging tools; a small selection of recent systems includes [9, 4, 25, 11, 2, 5, 26, 15, 16, 8]. We confine our survey of related work to research in specification languages, specification-based testing, and invariant inference systems.

7.1 Specification Languages

The basic concepts in our specification language (objects and relations) are the same as in object modeling languages such as UML [22] and Alloy [13], and the specification language itself has many of the same concepts and constructs as the constraint languages for these object modeling languages, which are designed, in part, to be easy for developers to use.

Standard object modeling approaches have traditionally been used to help developers express and explore high-level design properties. One of the potential benefits of our approach is that it may enable developers establish a checked connection between the high-level concepts in the model and their low-level realization in the data structures in the program.

7.2 Specification-Based Testing

Specification-based testing (of which Archie is an instance) tests the correctness of an execution by determining if it satisfies a specification written in some specification language. Specification-based testing is usually implemented at the granularity of procedure preconditions and postconditions. ADL [24], JML [14], Testera [18], Korat [3], and several Eiffel [19] implementations, to name a few, implement various forms of this kind of specification-based testing.

Archie, in contrast, implements a global invariant checker with no attempt to verify any property of the execution other than the preservation of the invariant. Advantages of Archie include reduced specification overhead and complete coverage of the global invariants (instead of checking more targeted properties that are intended to characterize procedure executions); the disadvantage is that it is not intended to find errors that do not violate the invariant. Our evaluation is that the two kinds of checkers address complementary properties and that both provide valuable checking functionality.

7.3 Invariant Inference and Checking

Several research groups have developed systems that dynamically infer likely invariants or other program properties; the same technology can be easily used to check the inferred properties (or, for that matter, any property expressed using the same formalism). Specific systems include DAIKON [10], Carrot [21], DIDUCE [12], and automatic role inference [6].

An important difference between Archie and these previously existing systems is that Archie is designed to check the substantially more sophisticated properties characteristic of complex linked data structures that must satisfy important structural constraints. The (in our view minimal) overhead is the need to provide a specification of these properties instead of automatically inferring the properties. And in fact, it would be feasible to use automatic property discovery tools to generate Archie consistency constraints or to obtain an initial set of properties that could be refined to obtain a more precise specification.

8. CONCLUSION

Error localization is a necessary prerequisite for correcting software errors and often the primary obstacle. Archie addresses this problem by accepting a specification of key data structure consistency properties, then automatically checking that the data structures satisfy these properties. The Archie checker can help developers quickly localize data structure corruption errors to the region of the execution between two subsequent calls to Archie.

Our set of optimizations enables the Archie compiler to generate checking code that executes more than efficiently enough to enable an effective check frequency and support its routine use in an interactive debugging environment. Moreover, the results from our case study indicate that developers can almost immediately use Archie to substantially improve their ability to localize and correct errors in a substantial software system. We believe that Archie therefore holds out the potential to substantially improve the ability of developers to first localize, then correct, data structure corruption errors.

Acknowledgements

This research was supported in part by DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-63513, NSF Grant CCR00-73513, NSF Grant CCR-0209075, NSF Grant CCR-0341620, and NSF Grant CCR-0325283,

9. REFERENCES

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

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103+, 2001.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[4] J.-D. Choi 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.

[5] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time, 2002.

[6] B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *ICSE02*, May 2002.

[7] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, October 2003.

[8] M. Ducass. Coca: An automated debugger for c. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[9] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *SOSP*, October 2003.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[11] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, pages 69–82, 2002.

[12] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, May 2002.

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

[14] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.

[15] R. Lencevicius, U. Hlzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA97*, October 1997.

[16] B. Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging AADEBUG 2003*, 2002.

[17] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[18] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, Nov. 2001.

[19] B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, NY, 1992.

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

[21] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, September 2003.

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

[23] B. D. Sanford et al. Center/tracon automation system: Development and evaluation in the field. In *38th Annual Air Traffic Control Association Conference Proceedings*, October 1993.

[24] S. Sankar and R. Hayes. Specifying and testing software components using ADL. Technical Report TR-94-23, Sun Microsystems, 1994.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[26] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, 2002.