Efficient Specification-Assisted Error Localization and Correction

Brian Demsky Cristian Cadar Daniel Roy Martin Rinard
Laboratory for Computer Science
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) improve the performance of the generated checking algorithm by over a factor of 3,900 as compared with the initial interpreted implementation, 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 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. These results illustrate Archie's potential to substantially improve current error localization and correction techniques.

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 distance between the error and its manifestation as observably incorrect behavior. The greater this distance, 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 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 errors in this system. These results illustrate Archie's potential to substantially improve current error localization and correction techniques.

[1]Archie is named after Archie Goodwin, the assistant to Rex Stout's fictional detective Nero Wolfe. The basic 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.

1.1 Specification-Based Approach

Archie accepts a specification of key data structure consistency properties (especially 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 the Archie consistency checker into the software system. If the system contains an error that corrupts the data structures, 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. Given these rules, Archie (conceptually) interprets the model definition rules to build an abstract model of the concrete data structures, then examines the model to find any violations of the consistency properties. The model itself operates at the level of abstract relations between abstract objects. 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 in the program, 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.

1.2 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 to frequent checking, however, is the overhead of executing the checks. Unfortunately, we found that our initial direct implementation of the consistency checking algorithm as described above was too inefficient for practical use. We therefore implemented the following optimizations:

- **Compilation:** The Archie compiler generates a C implementation of the Archie consistency checking algorithm, eliminating the interpretation overhead in the original Archie implementation.

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

- **Relation Elimination:** The compiler analyzes the specification to, when possible, replace the explicit construction of each relation 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 3,900 times faster on our benchmark software system than the original interpreted 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.

1.3 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 error localization 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 they encountered 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 alloted 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.

1.4 Contributions and Organization

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 designed to support error localization and correction.

- **Optimizations:** It presents a set of optimizations (compilation, 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 3,900, 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.

```
structure city {
  int population;
}
structure tile {
  int terrain;
  city *city;
}

tile grid[EDGE * EDGE];
```

Figure 1: Structure Definitions

The remainder of the paper is structured as follows. Section 2 presents an example that illustrates how Archie works, Section 3 presents the specification language, and Section 4 presents the Archie compiler and its optimizations. Section 5 discusses how we expect Archie to be used in practice, Section 6 presents the results of our case study, and Section 7 presents related work. We conclude in Section 8.

2. EXAMPLE

We next present an example (inspired by the FreeCiv program discussed in Section 6) that illustrates how Archie works. The program in question maintains a rectangular 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 for our example. There are separate structures for cities and tiles; the grid is an array of tiles.

Even a data structure this simple comes with 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.

- No city is placed on an ocean tile.

2.1 Expressing Consistency Properties

To express these constraints in our specification language, the developer first identifies the sets and relations in the conceptual data model that the concrete data structures implement. 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. In general, sets can contain primitive values such as integers or booleans and structures from the program. In our example, the `TILE` set contains `tile` structures, and the `CITY` set contains `city` structures. Each relation consists of a set of tuples chosen from two specified sets.

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

Figure 2: Object 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 for Example

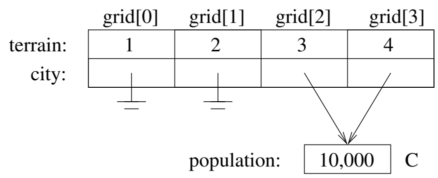

Figure 4: Concrete Data Structure

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

Figure 5: Model Constructed for Example

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 in the program 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 either an object that must be in a given set or a tuple that must be in a given relation. Archie processes these rules to construct the `TILE` and `CITY` sets and the `TERRAIN` and `CITYMAP` relations. Conceptually, the algorithm repeatedly finds a rule and a binding of the rule's quantified variables that satisfies the rule's guard. It then adds either the specified object into the specified set or the specified tuple into the specified relation to ensure that the inclusion constraint is satisfied. This algorithm continues until it reaches a fixed point. For the data structure instance 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 must be true for the constraint to be satisfied.

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 an ocean tile.[2] Note that the notation `CITYMAP.c` denotes the inverse image of `c` under the relation `CITYMAP` (the set of all `t` such that $\langle$`t,c`$\rangle$ in `CITYMAP`). As this example illustrates, the

[2]We use the C preprocessor to substitute out symbolic constants such as `MIN`, `MAX`, and `OCEAN`.

```
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 for Example

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 the Archie consistency checker. This checker examines the data structures and reports any inconsistencies to the developer, localizing the error that caused the inconsistency to the region of the execution between the failed check and the previous successful check. In our example, the consistency checker, when invoked on the data structure in Figure 4, would report that the structure violates the second rule in the specification. We allow the specification developer to include an explanatory comment for each rule; in addition to the violated rule, Archie also prints this explanation. In our example, the explanation might indicate that the second rule requires no city to have more than one location.

When the instrumented program executes, Archie localizes the error to the region of the execution between the failed call to the consistency checker and the last preceding successful call 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.

2.3 Optimizations

To increase the performance of the consistency checking, we implemented the following optimizations in the Archie compiler.

2.3.1 Fixed Point Elimination

In general, Archie may have to use a work-list-based fixed-point algorithm to compute the sets and relations in the model. But in some cases it may be possible to analyze the specification to generate a more efficient algorithm. The key is to find an efficient schedule for evaluating the model definition rules.

For the model definition rules in Figure 3, the first rule creates the `TILE` set, then the next several rules use the `TILE` set to create the `CITY` set and the `TERRAIN` and `CITYMAP` relations. The compiler can therefore generate efficient code that first traverses the `grid` array to construct the `TILE` set, then iterates through the `TILE` set to create the other sets and relations. The key property is that the compiler can order the rules so that the construction of the set or relation in one rule does not depend on any set or relation computed in any succeeding rule. In our example this order is simply the first rule followed (in any order) by the next several rules.

2.3.2 Relation Elimination

Archie's next optimization recognizes situations when it is possible to compute a relation on demand instead of eagerly constructing the relation, then uses the precomputed relation during constraint checking. In our example, the model construction rule that constructs the `TERRAIN` relation quantifies over all tiles `t` to insert $\langle$`t,t.terrain`$\rangle$ into the relation. Moreover, the consistency checking rules always use the `TERRAIN` relation in the forward direction — they always start with a tile `t` and constrain the value `t.TERRAIN` to which `TERRAIN` maps `t`. It is therefore possible to replace each use of the `TERRAIN` relation with code that computes `t.TERRAIN` instead of retrieving `t.TERRAIN` from a precomputed representation of the relation. This optimization eliminates both the space overhead of representing the relation and the time overhead of constructing and using the relation.

2.3.3 Set Elimination

The `TILE` set in our example is produced by a single model construction rule, then used (after relation elimination) only in quantifiers that iterate over all of the elements in the set. These quantifiers occur in three places: in the model construction rules in Figure 3 that create the `CITYMAP` relation and `CITY` set, and in the first consistency constraint in Figure 6, which checks that each tile has a legal terrain value. It is possible to replace the `TILE` set in the implementation of each of these quantifiers with a computation that efficiently enumerates all of the elements of the `TILE` set. This transformation eliminates the need to construct the `TILE` set, which in turn eliminates both the space overhead of representing the set and the time overhead of constructing the set.

2.3.4 Optimized Execution

After optimization, the consistency checker executes as follows. It first iterates over the tiles in the grid to check that that every tile has a legal terrain value and to construct the `CITYMAP` relation and the `CITY` set. It then iterates over the `CITY` set and uses the `CITYMAP` relation to check the last two consistency constraints in Figure 6. This optimized implementation replaces the construction of the `TILE` set and `TERRAIN` relation with efficient computations distributed throughout the generated code. Together, all of these optimizations reduce the overall execution time of the consistency checking in our benchmark software system by a over factor of 3,900. Because the `TILE` set and `TERRAIN` relation are significantly larger than the `CITY` set and `CITYMAP` relation, they also substantially reduce the memory requirements.

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 allows the developer to declare the layout of the data structures. Figure 7 presents the grammar for this language. 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 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.

The structure definition language is similar to that of C. However, it supports wider range of primitive data types, provides a form of structure inheritance, and allows the developer to define in-line, variable-length arrays. These extensions enable the developer to precisely specify the format of the elements in many heavily encoded data structures.

$$
\begin{array}{rcl}
structdefn & := & \texttt{struct}\ structurename \\
 & & (subtypes\ structurename)\ \{fielddefn^*\} \\
fielddefn & := & type\ field; \mid \texttt{reserved}\ type; \mid \\
 & & type\ field[E]; \mid \texttt{reserved}\ type[E]; \\
type & := & \texttt{boolean} \mid \texttt{byte} \mid \texttt{short} \mid \texttt{int} \mid \\
 & & structurename \mid structurename\ * \\
E & := & V \mid number \mid string \mid E.field \mid E.field[E] \mid \\
 & & E - E \mid E + E \mid E/E \mid E * E
\end{array}
$$

Figure 7: Structure Definition Language

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

Figure 8: Model Definition Language

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` $S_1, ..., S_n$ declares a set `S` that contains objects of type `T`, where `T` is either a primitive type (with the range optionally constrained to be between two given values) or a `struct` type declared in the structure definition part of the specification. The set `S` has `n` subsets $S_1, ..., S_n$ which together partition `S`. Changing the `partition` keyword to `subsets` removes the requirement that the subsets $S_1, ..., S_n$ partition `S` but otherwise leaves the meaning of the declaration unchanged. A relation declaration of the form `relation R:` S_1`->`S_2 specifies a relation between the objects in the sets S_1 and 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 8 presents the grammar for the 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.

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

3.3 The Constraint Language

Figure 9 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.

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

Figure 9: Model Constraint Language

4. COMPILATION AND OPTIMIZATION

Our initial implementation of Archie interpreted the specification every time it performed a consistency check. We found that this implementation was too slow to satisfy our needs; in particular, it increased the running time of our benchmark software system by almost a factor of 25,000. To eliminate the interpretation overhead, we developed a compiler that processed the Archie specification to generate C code that implemented a basic consistency checking algorithm. This algorithm first constructs each of the sets and relations in the model, then evaluates the consistency constraints to detect any possible inconsistencies. It uses a work-list-based fixed-point algorithm to ensure that it correctly constructs the model. While this baseline compiled version executes almost five times faster than the interpreted version, it was still 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 $\texttt{for}\ V\ \texttt{in}\ S$ (or of the form $\texttt{for}\ \langle V_1, V_2 \rangle\ \texttt{in}\ R$) or if the rule has a guard of the form $E\ \texttt{in}\ S$ (or $\langle E_1, E_2 \rangle\ \texttt{in}\ R$). A rule *defines* a set S (or relation R) if it has a inclusion constraint I of the form $E\ \texttt{in}\ S$ (or $\langle E_1, E_2 \rangle\ \texttt{in}\ R$).

The compiler finds the strongly connected components in the dependence graph and topologically sorts these components. 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 uses a work list to implement 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 consistency constraint 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 $Q^*,\ G_1 \wedge G_2[V_2/V_1] \Rightarrow I[V_2/V_1]$. 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 $Q^*,\ G \Rightarrow C[V_2/V_1]$. To apply the transformation, the model definition rule must be the only rule that defines S. Note that the new constraint has a predicate $(G \Rightarrow C[V_2/V_1])$ 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 required to compute the set and the space required to 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 in the model.

4.4 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.

Version	Time
No Instrumentation	0.234 sec
Interpreted	95 min
Baseline Compiled	20 min
Fixed point elimination	25.60 sec
Relation Elimination	10.66 sec
Set removal	1.45 sec

Table 1: Performance Results

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

- **Black Box Usage:** We expect the Archie specifications to be developed by a small number of developers who are comfortable using the specification language. The remainder of the developers can simply use Archie as a black box by invoking the Archie consistency checker. We anticipate no need for the vast majority of the developers to learn the Archie specification language or to become comfortable using it. There is also no need to change the programming language,[3] coding style, or other development tools.

- **Incremental Adoption:** Archie supports an incremental adoption strategy — the developer can start with a specification that captures a small subset of the consistency properties, then incrementally augment the specification to capture more and more properties. During the entire specification development process the consistency checker remains operational and increasingly useful as more properties are added. Calls to the Archie consistency checker 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.

- **Utility:** Based on the results of our case study in Section 6, we believe that developers will find Archie to be very useful in helping to localize and correct errors, and will therefore be motivated to use it as they develop and maintain their software.

- **Ease of Development:** Based on our experience developing similar specifications in another project [8], we believe that Archie specifications will prove to be relatively easy to develop once the developer understands the relevant data structures.[4] 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 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

To evaluate the effectiveness of our tool, we obtained a benchmark software system, 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 debugging effectiveness of the developers that used Archie with the developers that did not, we are able to obtain an indication of how well Archie supported the debugging process for this system, and, by extension, for other systems as well.

6.1 Developer Population

We recruited six developers with relatively homogeneous backgrounds: all developers were born and educated through high school in Romania or Moldova and all represented their home country in international programming competitions while they were in high school. All of the developers are currently either undergraduate or graduate 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. This error caused the `heapify` operation [5] to incorrectly swap the value of the parent node with the value of its largest child

[3] The current Archie compiler generates C code and is designed to work with programs written in C and C++. It is straightforward to retarget Archie to generate code for other programming languages.

[4] Specifically, we have developed specifications for the FreeCiv interactive game discussed in Section 6, the CTAS air-traffic control system [1, 20] (this deployed system consists of over 1 million lines of C and C++ code), a simplified version of the Linux ext2 file system [17], and the data structures in Microsoft Word files. With the exception of CTAS, we were able to develop all of our specifications in the course of a single week. The CTAS specification took another week, with much of the effort devoted (with the help of the CTAS developers) to understanding the CTAS data structures.

even though the value of the parent was larger than the value of that child. 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 74 .h and 68 .c files. It contains four modules: a server module, a client module, an AI module, and a common module that contains procedures called by the other three modules. 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 irrigation or 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 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.
- 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. 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 in the source code 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.

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)

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.

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 replaced the new (incorrect) version with the correct version that he had examined while searching for the second error!

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 long distance 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.[5] As these results show, once the NoTool developers were given access to Archie, they were able to quickly localize and correct the errors.

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

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

Our evaluation is that error localization was the crucial step for debugging the errors in our study and that 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.

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 only 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 over an hour and half. 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. For this we need a much more efficient implementation.

Our optimizations enabled us to provide the developers in our study with a checker that can execute frequently (enabling excellent error localization) 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.

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, 22, 11, 2, 6]. We confine our survey of related work to research in specification languages, specification-based testing, hand-coded property checkers, 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 [19] 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 [21], JML [14], Testera [15], Korat [3], and several Eiffel [16] 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. In particular, it does not attempt to verify that the procedure satisfies its postcondition. Advantages of Archie include reduced specification overhead and complete coverage of the global invariant (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 Hand-Coded Property Checkers

It is possible to directly implement checking algorithms in the same programming language as the rest of the software system. In fact, we have developed such checkers ourselves and believe that others have as well. One potential advantage of this approach is the ability to hand-optimize the algorithm to minimize the checking overhead; disadvantages include the need to develop and order the data structure traversal algorithms and to implement any auxiliary data structures required to check the desired property. Developing this code can be especially difficult because the developer cannot assume that the input data structures satisfy *any* property — the whole point of the checker is to detect data structures that may (arbitrarily) violate their invariants.

In our experience hand-coded consistency checkers are vulnerable to anomalies such as infinite traversal loops, incomplete property coverage, errors caused by unwarranted assumptions about the input data structures and, in comparison with specification-based approaches, increased development overhead. Nevertheless, we believe that the widespread use of such hand-coded checkers would be an improvement over current practice.

In an attempt to better understand the issues, we developed several hand-coded consistency checkers for the FreeCiv software system. These checkers were substantially larger and more difficult to develop than our FreeCiv specifications. They were also comparably as efficient as (but not significantly more efficient than) our most heavily optimized Archie checkers.

7.4 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 [18], DIDUCE [12], and automatic role inference [7].

An important difference between Archie and these previously existing systems is that Archie is designed to check 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 to eliminating the error. Archie addresses this problem by accepting a specification of key data structure consistency properties, then automatically checking that the data structures satisfy these properties. By inserting calls to the Archie checker into their software system, developers can localize data structure corruption errors to the region of the execution between the call that detects the corrupt data structure and the previous call, which verified that the data structures were consistent.

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.

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. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs, 2002.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press/McGraw Hill, 2001.

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

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

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

[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] 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.

[16] B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, NY, 1 edition, 1992.

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

[18] 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.

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

[20] 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.

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

[22] 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.