ISR Institute for Software Research
University of California, Irvine

Static Analysis of Task Interactions in Bristlecone for Program Understanding

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

Sivaji Sundaramurthy
University of California, Irvine
srsundar@uci.edu

October 2007

ISR Technical Report # UCI-ISR-07-7

Institute for Software Research
ICS2 217
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

Static Analysis of Task Interactions in Bristlecone for Program Understanding

Brian Demsky and Sivaji Sundaramurthy
Institute for Software Research
University of California, Irvine
{bdemsky, srsundar}@uci.edu

ISR Technical Report # UCI-ISR-07-7

October 2007

Abstract

We have developed a static analysis to help developers understand the interactions between objects and tasks in Bristlecone applications. The Bristlecone language was designed to help developers construct robust applications out of potentially unreliable components. Bristlecone applications can adapt their behavior, potentially degrading their functionality, in response to software errors in order to avoid catastrophic failures.

Bristlecone applications are composed of a set of tasks. The description of the behavior of these tasks is split into two orthogonal specifications: a set of high-level task specifications that describe both when the task should be invoked and which objects the task operates on and low-level imperative specifications that describe the operational behavior of the task.

The Bristlecone compiler and runtime use the high-level specifications to detect software errors, to recover the application from an error to a consistent state, and to reason how to safely continue the application's execution after the error.

This paper presents a static analysis that automatically extracts information about the interaction of objects and tasks, a set of graphical representations that capture the relevant information about these interactions, and a web-based, interactive tool that uses these graphical representations to communicate this information to the developer. We have used this tool to explore the behavior of several benchmark applications including an online game, a web server, and a chat server. Our experience indicates that these analysis results are useful for understanding the interaction between tasks and objects in our benchmark applications and correcting several software bugs.

Static Analysis of Task Interactions in Bristlecone for Program Understanding

Brian Demsky and Sivaji Sundaramurthy
Institute for Software Research

ISR Technical Report # UCI-ISR-07-7
October 2007

***Abstract*— We have developed a static analysis to help developers understand the interactions between objects and tasks in Bristlecone applications. The Bristlecone language was designed to help developers construct robust applications out of potentially unreliable components. Bristlecone applications can adapt their behavior, potentially degrading their functionality, in response to software errors in order to avoid catastrophic failures.**

Bristlecone applications are composed of a set of tasks. The description of the behavior of these tasks is split into two orthogonal specifications: a set of high-level task specifications that describe both when the task should be invoked and which objects the task operates on and low-level imperative specifications that describe the operational behavior of the task.

The Bristlecone compiler and runtime use the high-level specifications to detect software errors, to recover the application from an error to a consistent state, and to reason how to safely continue the application's execution after the error.

This paper presents a static analysis that automatically extracts information about the interaction of objects and tasks, a set of graphical representations that capture the relevant information about these interactions, and a web-based, interactive tool that uses these graphical representations to communicate this information to the developer. We have used this tool to explore the behavior of several benchmark applications including an online game, a web server, and a chat server. Our experience indicates that these analysis results are useful for understanding the interaction between tasks and objects in our benchmark applications and correcting several software bugs.

***Index Terms*— Program Understanding, Static Analysis**

I. Introduction

Software faults pose a significant challenge to constructing robust software systems. The current approach to addressing this problem is to work hard to minimize the number of software faults in software systems through a combination of development processes, automated tools, and testing. While minimizing the number of software faults is a critical component of the development process for reliable software, it is not sufficient: some software faults will inevitably slip through the development and testing processes.

Some software systems, including the Lucent 5ESS telephone switch [24] and the IBM MVS operating system [33], utilize hand-developed recovery routines to improve reliability in the face of these remaining software faults. These routines detect and repair software errors enabling the software systems to successfully recover. Many programming languages, including Java and C++, contain built-in exception handling primitives that are designed to facilitate writing recovery code. Unfortunately, this approach requires the developer to correctly predict what types of software faults are likely to occur in practice and to reason about how the software system can recover from the errors produced by the execution of these faults.

We have previously developed Bristlecone, a new programming language for robust software systems, to enable developers to construct robust software systems out of unreliable components [13]. Our previous research indicated that Bristlecone significantly improved the robustness of several benchmark applications to randomly injected failures.

Bristlecone programs consist of a set of decoupled *tasks* with each task encapsulating an individual conceptual operation and a set of task specifications that describe how these decoupled tasks interact. Bristlecone tracks the abstract state of objects using object *flags* and tracks groups of related objects using *tag* instances. The key idea is that if the Bristlecone runtime detects an software error, the runtime can use this extra information about the structure of the program and abstract object states to isolate any effects of the error and then to adapt the execution of the software system to enable the program to continue to execute past the error. The Bristlecone runtime detects errors through a combination of data structure consistency checks [12] and monitoring for illegal operations such as illegal memory accesses, library usage errors, or arithmetic errors.

This paper presents a static analysis for Bristlecone that automatically extracts a model of the interaction between the tasks and objects in a Bristlecone application and produces a set of graphical abstractions that communicate the extracted information. We have implemented an interactive, graphical tool that presents this information in an intuitive form to the developer.

A. Contributions

This paper makes the following contributions:

- **Static Analysis of Task Interaction Language:** It presents a static analysis of the task specifications that extracts a set of reachable abstract object states, the initial abstract states for any objects that a task allocates, and a set of transitions between abstract objects states that model the effects of the tasks.

- **Graphical Representations:** It presents the two graphical representations that our tool uses to communicate the results of the analysis to the developer. The first graphical representation is the *flag state transition diagram*, a graph in which the nodes represent the possible abstract states of an object's tags and flags and the edges represent the changes the tasks induce on the abstract object states. This graph is intended to help developers visualize the interaction patterns between tasks and objects — including both how tasks affect the states of an object's flags and tags and how these changes enable other tasks to operate on these objects.
  The second graphical representation is the *task diagram*, a graph in which the nodes represent tasks and the edges represent whether a second task can be invoked on an object immediately after the first task exits. This graph is intended to help the developer understand how objects flow between tasks.
- **Graphical Exploration Tool:** It presents an interactive, graphical interface that allows the developer to explore the results of the flag state analysis.
- **Experience:** It presents our experiences using the tool to both understand the behavior of benchmark programs and to identify and correct software faults in a buggy program.

II. Example

We next present a web server example. This web server contains specialized e-commerce functionality and maintains state to track an online store's inventory.

As the example web server executes, it creates, modifies, and destroys objects. During an individual object's lifespan, the conceptual state or role of that object in the larger computation may evolve. This evolution may change the way that the software system uses the object or change the functionality of the object. For example, the Java `connect` method changes the functionality of a `Socket` object in a computation: after the `connect` method is invoked, data can be written to or read from that `Socket` object.

The Bristlecone language provides the flag construct, which the developer can use to track the conceptual state of an object. The runtime uses the conceptual state of the object as indicated by the object's flag to determine which conceptual operations or tasks to invoke on the given object. When a task exits, it can change the status of the flags of its parameter objects.

Figure 1 presents part of the `WebConnection` class definition. The `WebConnection` class definition declares three flags: the `initialized` flag, which indicates whether the `WebConnection` class is in its initial state; the `file_req` flag, which indicates that the server has received a file request from this client connection; and the `write_log` flag, which indicates whether the connection information is available for logging.

In many cases, the developer may need to invoke a task on multiple objects that are related in some way. The Bristlecone language provides a tag construct, which the developer can use to group objects together. New instances of

```
class WebConnection {
  /* This flag indicates that the WebConnection
     object is in its initial state. */
  flag initialized;

  /* This flag indicates that the system has
     received a request to send a requested
     file. */
  flag file_req;

  /* This flag indicates that the connection
     should be logged. */
  flag write_log;
  ...
}
```

Fig. 1. WebConnection Class Declaration

tags are created using tag allocation statements of the form `tag tagname=new tag(tagtype)`. Such a tag allocation statement allocates a new tag instance of type `tagtype` and assigns the variable `tagname` to this tag instance. The developer can tag multiple objects with a tag instance to group them, and then use that tag instance to ensure that the runtime invokes a task on two or more objects in the group defined by the tag instance. For example, the web server uses tags to group a `WebConnection` object with the corresponding `Socket` object that provides the TCP connection for that web request. Tag instances can be added to objects when the object is allocated, and they can be added or removed to or from a task's parameter objects when the task exits.

A. Tasks

Bristlecone software systems consist of a collection of interacting tasks. The key difference between tasks and methods is that the runtime directly invokes a task when the heap contains objects with the specified flags and tags to serve as the task's parameters while method invocation is performed by tasks or other methods. The runtime uses a task's specification to both determine which objects serve as the task's parameters and when to invoke the task. A second difference is that unlike methods, tasks are not declared as part of any class.

Each task declaration consists of the keyword `task`, the task's name, the task's parameters, and the body of the task. Figure 2 presents the task declarations for the web server example. The first task declaration in Figure 2 declares a task named `startup` that takes a `StartupObject` object as a parameter and points the parameter variable `start` at this object. The declaration also states that the `StartupObject` object must have its `initialstate` flag set before the runtime can invoke this task. The runtime uses this flag information to determine when it can legally invoke this task. Before exiting, the `taskexit` statement in the `startup` task resets the `initialstate` flag in the `StartupObject` object to false to prevent the runtime from repeatedly invoking the `startup` task.

B. Normal Execution of the Web Server

During normal execution, the web server performs the following operations (although not necessarily in this order):

```
/* This task starts the web server */
task startup(StartupObject start in
     initialstate) {
  ...
  ServerSocket ss=new ServerSocket(80);
  Logger l=new Logger() (set initialized to true);
  taskexit((start: set initialstate to false));
}

/* This task accepts incoming connection
   requests and creates a Socket object. */
task acceptConnection(ServerSocket ss in
     pending_socket) {
  ...
  tag t=new tag(connection);
  WebConnection w=new WebConnection(...)
    (set initialized to true)(add t);
  ss.accept(t);
  ...
}

/* This task reads a request from a client. */
task readRequest(WebConnection w in initialized
     with connection t, Socket s in IO_Pending
     with connection t) {
  ...
  taskexit((w: set initialized to false,
    set file_req to true, set write_log to
      true));
}

/* This task sends the request to the
   client. */
task sendPage(WebConnection w in file_req
     with connection t, Socket s with
     connection t) {
  ...
  taskexit((w: set file_req to false));
}

/* This task logs the request. */
task logRequest(WebConnection s in write_log,
  Logger l in initialized) {
  ...
  taskexit((s: set write_log to false));
}
```

Fig. 2. Task Specifications

- **Accepting Connections:** At some point, the web server will receive an incoming connection request from a web browser. This will cause the runtime to set the `ServerSocket` object's `pending_socket` flag to true, which will in turn cause the runtime to invoke the `acceptConnection` task with this `ServerSocket` object as its parameter.
  The `acceptConnection` task allocates a `WebConnection` object to maintain state associated with this web request and a `Socket` object to manage the underlying network connection. This task also creates a new `connection` tag instance and then uses this tag instance to group the `WebConnection` object with the `Socket` object.
- **Receiving Requests:** After the connection is established, the client web browser sends a web page request to the server. In response to this incoming web page request, the runtime sets the `Socket` object's `IO_pending` flag to true, which in turn. causes the runtime to invoke the `readRequest` task.
  If the `readRequest` task has received the complete request, it sets both the `file_req` flag and the `write_log` flag to true and resets the `initialized` flag to false. These flag changes cause the runtime to eventually invoke both the `sendPage` and the `logRequest` tasks and prevents repeated invocations of the `readRequest` task on the same objects.
- **Serving Requests:** The `sendPage` task then reads the requested file and sends the contents of the file to the client browser. The `sendPage` task then resets the `received_request` flag to false to prevent repeated invocations of the `sendPage` task.
- **Logging Requests:** Finally, the `logRequest` task writes a log entry to record which web page was requested. The `logRequest` task then resets the `write_log` flag to false to prevent repeated invocations of the `logRequest` task.

C. Flag State Transition Diagrams

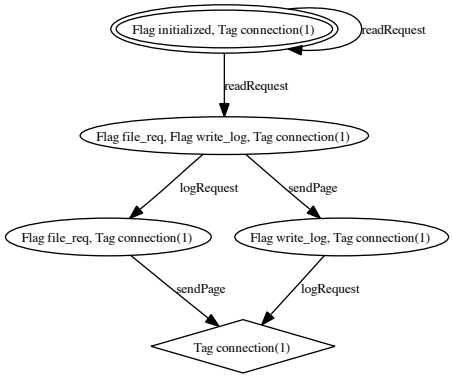

Fig. 3. Flag State Transition Diagram for the `WebConnection` class

Figure 3 presents a diagram of the dependences between tasks in the web server example. The nodes in this diagram represent the abstract flag states of objects and the edges represent transitions between these flag states that the tasks perform. Each abstract flag state specifies the truth value assignments for an object's flags and an abstraction of the number of tag instances of a given type bound to the object. The double periphery of the node labeled `Flag initialized, Tag connection(1)` indicates that newly allocated objects can be created with this flag state. The node's label indicates that these objects have their `initialized` flag set to true and have been tagged with exactly one `connection` tag instance. The edges labeled `readRequest` from this node model the actions of the `readRequest` task on the flag state of objects. The self-edge labeled `readRequest` indicates that it is possible for the `readRequest` task to leave a `WebConnection` object in its initial state. This case occurs if the web server has

only received a partial web page request. The other edge labeled `readRequest` indicates that the `readRequest` task can cause a `WebConnection` object to transition from the initial state into the state labeled `Flag file_req, Flag write_log, Tag connection(1)`. This case occurs when the web server has received the complete web page request.

The diamond shape of the node labeled `Tag connection(1)` indicates that no task can fire on this object, and therefore, that this object will be garbage collected unless a live object references it. The elliptical shapes of the remaining nodes indicate that objects in these flag states can transition to a garbage collectible state. (A rectangular node would indicate that objects cannot transition to a garbage collectible state, and therefore can never be garbage collected.)

D. Task Diagrams

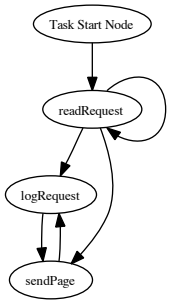

Fig. 4. Task Diagram for the `WebConnection` class

Figure 4 gives the task diagram for the web server example. The nodes in this diagram represent the tasks that operate on the `WebConnection` object. The edges model the flow of objects between tasks — there is an edge from one task to a second task if the first task exits placing its parameter object in a flag state that can trigger the second task. From this diagram we can observe that the web server must first execute the `readRequest` task before it can execute either the `logRequest` or `sendPage` tasks.

E. Understanding Consequences of Failures

Bristlecone is designed to enable software systems to automatically recover from failures to successfully continue execution. To ensure that a failed task cannot leave a data structure in a partially updated state or otherwise corrupt the data structure, the Bristlecone runtime encloses the invocation of a task in a transaction. The classic problem with using transactions to recover from failures is that after the transaction restores the program's state, the program will repeatedly fail in the same way and never execute beyond the deterministic failure. Bristlecone solves this problem by using the task specifications to avoid repeating the same failure — the runtime uses the task specifications to determine which tasks other than the failed task can be safely executed. The task specifications were designed to enable the runtime to determine whether data structures are in states that tasks can safely operate on.

$flagdecl$	:=	`flag` $flagname$; | `external flag` $flagname$;
$tagdecl$	:=	$tagtype$ $tagname$;
$taskdecl$	:=	`task` $name(taskparamlist)$
$taskparamlist$	:=	$taskparamlist, taskparam$ | $taskparam$
$taskparam$	:=	$type$ $name$ `in` $flagexp$ | $type$ $name$ `in` $flagexp$ `with` $tagexp$;
$flagexp$	:=	$flagexp$ `and` $flagexp$ | $flagexp$ `or` $flagexp$ | !$flagexp$ | ($flagexp$) | $flagname$ | `true`
$tagexp$	:=	$tagexp$, $tagtype$ $tagname$ | $tagtype$ $tagname$
$statements$	:=	... | `taskexit`($flagactionlist$) | `tag` $tagname$ = `new tag`($tagtype$) | `new` $name(params)(flagactions)$ | `new` $name(params)(tagactions)$ | `new` $name(params)(flagactions)(tagactions)$
$flagactionlist$	:=	$flagactionlist, varflagaction$ | $varflagaction$
$params$	:=	... | `tag` $tagname$
$varflagaction$	:=	($name$: $flagactions$)
$flagactions$	:=	$flagactions, flagaction$ | $flagaction$
$flagaction$	:=	`set` $flagname$ `to` $bool$
$tagactions$	:=	$tagactions, tagaction$ | $tagaction$
$tagaction$	:=	`add` $tagname$ | `clear` $tagname$
$flagname$	:=	$name$
$bool$	:=	`true` | `false`
$assertionlist$	:=	$assertionlist, assertion$ | $assertion$
$assertion$	:=	$specificationname(bindinglist)$
$bindinglist$	:=	$bindinglist, binding$ | $binding$
$binding$	:=	$var : expression$

Fig. 5. Task Grammar

In many cases, developers may wish to explore the possible consequences of a task failure. This can be useful for deciding which tasks are more critical than others and, therefore, should receive more of the limited development resources. Developers can use the flag transition diagrams to understand the consequences of task failures. For example, we can observe from Figure 3 that failures of `readRequest` task can prevent both the `logRequest` and `sendPage` tasks from executing. We can also see that the `logRequest` and `sendPage` tasks are mutually failure independent — if the execution of one of these task fails, the runtime system will still execute the other task by abort the first task thereby returning to the fork in the graph and then choosing an alternate path at that fork.

III. Language Design

The Bristlecone language includes a task specification language to describe how to orchestrate task execution. We intend that the developer will construct software systems as collections of loosely coupled tasks. Bristlecone introduces object flags to store the conceptual state of the object and tags to group related objects. Each task contains a corresponding task specification that describes which objects the task operates on, when the task should execute, and how the task affects the conceptual state of objects.

Bristlecone is an object-oriented, type-safe language similar to Java. Bristlecone includes standard object-oriented functionality such as virtual dispatch. Figure 5 presents the grammar

for Bristlecone's task extensions. The developer includes a flag declaration inside a class declaration to declare that objects of that class contain the declared flag. Flag declarations use the `flag` keyword followed by the flag's name. The developer may optionally use the `external` keyword to specify that the flag is set and reset by the runtime system. External flags are intended to enable a software system to handle asynchronous events such as communication over the Internet or mouse clicks. External flags are intended to be declared in library code and the corresponding runtime component that provides support for that external event sets and clears the external flag.

The developer uses tags to enforce relations between the parameters of a task. The developer can create new tags with the `new tag` statement and a tag type. Note that there may be many instances of a given type of tag. Each different instance of a tag is distinct — objects labeled by two different instances of the same tag type are not grouped together.The developer can add tags to objects when an object is allocated or add or remove tags to or from parameter objects at the exit of a task.

The developer declares a task using the `task` keyword followed by the task's name, the task's parameters, and the task's code. Each task parameter declaration contains the parameter's name, the parameter's type, a flag guard expression that specifies the state of the parameter's flags, and an (optional) tag guard expression that specifies the tags the object has. The task may be executed when all of its parameters are available. A parameter is available if the heap contains an object of the appropriate type, that object's flags satisfy the parameter's guard expression, and that object contains any tag instances that the parameter's guard expression specifies. The Bristlecone language adds a modified `new` statement that specifies the initial flag and tag settings for a newly allocated object. These settings take effect when the task exits.

The Bristlecone language contains a `taskexit` statement that specifies changes to the states of the flags and tags of the parameter objects to be performed when the task exits. Note that a single task can have multiple `taskexit` statements — one for each possible exit point for the task.

IV. Static Analysis

The static analysis produces a flag state transition diagram for each class. The nodes in this diagram represent the possible flag states of an object: the flag state includes the boolean values of each flag in the class and an abstraction of the tag instances the object has been tagged with. Recall that an object can have many different tag instances of the same type. Therefore, the flag state abstracts the tag state with a 1-limited abstraction for tags; for each tag type, the flag state indicates whether that object has 0, 1, or at least 1 instance of that tag.

The edges in this diagram represent the possible invocations of tasks on parameter objects and the effect that these invocations have on the flag states of the parameter objects. If task T can be invoked with its ith parameter object in the flag state fs_1 and exits with this object in the flag state fs_2, then there is an edge for a task T for its ith parameter from the flag state node for fs_1 to the flag state node for fs_2, The lack of an edge implies that a transition is prohibited.

We have implemented the static analysis using two stages. The first stage computes: (1) the objects that each task allocates and (2) the initial flag states of these objects. The second stage uses the results of the first stage to compute: (1) the set of reachable states for each class and (2) which tasks can cause transitions between these states.

A. New Object Analysis

The first stage of the analysis determines the initial states of all objects that a task can potentially allocate. The complication is that Bristlecone tag parameters were designed to be polymorphic in that type of the tag to provide developers with the flexibility to pass any type of tag into a method that takes tag parameters. This flexibility is desirable as a developer may often need to group arbitrary combinations of objects including objects developed by other developers. However, as a result of this complication, the new object analysis must consider a method's calling context before it can determine the types of the tag instances that the method may use to tag newly allocated objects.

1) Add all tasks to queue.
2) Remove item q from head of the queue.
3) Clear the map from tag variables to tag types.
4) If q is a task, use the task specifications to add bindings from tag parameter variables to tag types to the map.
5) If q is a method with a list of tag parameter types, add these tag bindings to the map.
6) For each tag allocation in q, add a binding from the tag name to its tag type to the map.
7) For each method call in q, use the map to generate a list of bindings for any tag parameters in q and add this list and any method this call could potentially invoke to the queue if it has not been seen yet. Store that q calls this combination of method and tag binding.
8) For each allocation site in q, use the map to compute the types of tags the allocation has. Combine this information with the flag status changes for this allocation site to generate a newly allocated object state. Add this to q's set of allocations.
9) If queue is not empty goto step 2.
10) Visit method and tag binding pairs in a topologically sorted order. For each method and tag binding pair, compute all of the object flag states that it transitively allocates.

Fig. 6. New Allocation Analysis

Figure 6 presents pseudo-code for the new allocation static analysis. This algorithm starts with the set of tasks as its roots and explores the call graph, specializing each method with a calling context containing a list of the types of all its tag parameters. When the analysis processes a method or task it uses the method's calling context or the task's specification along with any tag declarations to determine the types of all tag variables. It can then process each allocation site to determine that allocation site's exact abstract flag state. When the analysis discovers a method call to a previously unseen combination of method and calling context, it adds that combination of method and calling context to the queue. It continues this process until it has processed all reachable tasks and method calling contexts. It then processes the tasks and method calling contexts in topological order to compute all of the flag states that either the task, the method, or any method that it (transitively) calls allocates.

1) Add `StartupObject` with its `initialstate` flag set to true to queue.
2) Remove flag state q from head of the queue.
3) Loop through each task t and each parameter i of task t.
4) Check if q's type is compatible with the declared type of parameter i of task t. If not, then goto step 3.
5) Check if q's flag state satisfies the flag guards of parameter i of task t. If not, then goto step 3.
6) Check if q's flag state has all of the tags declared for parameter i of task t. If not, then goto step 3.
7) Use the results of the new object allocation analysis to determine which new flag states task t can create. If the state transition analysis has not previously discovered this flag state, add the flag state into the queue.
8) Loop through each task exit in t. Apply the flag and tag changes described in t to the flag state q to generate the new flag state q'. Add an edge from q to q' and label this edge with the task t and parameter i. If the analysis has not previously discovered flag state q' yet, add the flag state q' to the queue.
9) If the queue is not empty, goto step 2.

Fig. 7. State Transition Analysis

B. State Transition Analysis

The state transition analysis determines the flag state transitions that tasks can potentially perform. This analysis uses the results of the new object analysis to determine the flag states of all objects that a task could potentially allocate.

Figure 7 presents pseudo-code for the state transition analysis. The analysis starts by processing the `StartupObject` object in its initial state with its `initialstate` flag set to true. For each flag state, the analysis analyzes the task specification to determine all of the tasks that an object with this flag state could potentially serve as a parameter. For each such task, the state transition analysis uses the results of the new object analysis to determine the flag states of all of the objects that this task may potentially allocate. For any flag state the analysis has not already discovered, it adds that flag state to the queue. The analysis then examines each possible task exit to determine how that task changes the flags and tags of the parameter object.

Our implementation extends this basic algorithm to conservatively analyze the action of the runtime on external flags by modelling the action of the runtime on an external flag as equivalent to a pair of tasks: one task that operates on all objects with the external flag cleared and sets it and a second task that operates on objects with the external flag set and clears it.

C. Automated Analysis of Flag State Transition Diagrams

We believe that automated analyses of the flag state transition diagrams can help the developer quickly understand key properties of an application's use of a certain class of object. We have developed an analysis of flag state diagrams that checks necessary conditions for an object to be garbage collected. In general, objects in Bristlecone can be garbage collected if (1) the object is unreachable from any potential parameter objects and (2) the object cannot be a parameter object of any task. This analysis determines which states can be garbage collected, from which states that objects can eventually transition into a garbage collectible state, and from which states objects can never reach a garbage collectible state. Our tool communicates this information to the developer through the shape of the nodes in the flag state transition diagram.

D. Task Diagram

Depending on the task at hand, the developer may wish to view a coarser abstraction of the program. Our tool can generate task diagrams to help the developer understand how objects flow between tasks. Task diagrams provide the developer with a task-centric view of the program. There is a task diagram for each class in the heap.

The nodes in a class's task diagram represent the tasks that take objects of that class as parameters. The edges in the diagram model the flow of objects between tasks — there is an edge from one task to a second task if a parameter object of the first task can be used as a parameter object of the second task immediately after the first task exits.

In some cases, the developer may wish to view how all of the tasks in the program interact. Our tool can generate an overview task diagram that unifies all of the class task diagrams into a single diagram. In addition, overview task diagrams contain edges that capture the dependence between a task that allocates new objects and other tasks that operate on these newly allocated objects. This diagram gives the developer an overview of the relationships between all of the tasks in an application.

V. User Interface

The user interface presents five kinds of web pages: flag state transition diagram pages, flag state allocation pages, task pages, task diagram pages, and an overview task diagram page. Each flag state transition page presents the flag state transition diagram for a class. From these pages, the developer can click on both nodes with double peripheries and edges to see the corresponding flag state allocation pages and task pages, respectively. Each flag state allocation page presents the set of tasks that may allocate new objects with the corresponding initial flag state. The developer can click on these tasks to bring up the corresponding task page. Each task page presents a list of the task's parameters and a list of the initial flag states for all objects that the task allocates. This web page contains a link for each parameter and each newly allocated flag state to the corresponding flag state transition diagram page and task diagram page. Each task diagram page presents the task diagram for a class. The developer can click on a task node in this diagram to see the corresponding task page. Finally, the overview task diagram page presents a task diagram for the entire application. The developer can click on task node in this diagram to see the corresponding task page.

VI. Experience

We next discuss our experiences using the flag state analysis tool to explore the behavior of several Bristlecone programs. We have implemented the Bristlecone compiler. Our implementation consists of approximately 19,700 lines of Java

code and C code for the Bristlecone compiler and runtime system. The Bristlecone compiler generates C code that runs on both Linux and Mac OS X. The Bristlecone runtime uses a precise stop-and-copy garbage collector. The source code for our compiler and runtime including the static analysis and web interface is available at `http://newport.eecs.uci.edu/~bdemsky/bristlecone/`. We report our experience for: TTT, a tic-tac-toe game; a web server; and a chat server. In these experiments, the individual using the static analysis tool had no prior experience with the benchmark program.

A. Tic-Tac-Toe Server

TTT, a tic-tac-toe game server, was developed by a student in the author's class as a class project. Users can connect to TTT through telnet and play a game of tic-tac-toe against the computer. This was the student's first experience with Bristlecone. The student only had access to example programs and the Bristlecone technical report — the student did not have access to a Bristlecone tutorial or receive any other assistance writing TTT. When we attempted to run TTT, we discovered some surprising behaviors. For example, the server did not allow us to complete the game and it printed out multiple copies of the board after each move. We used the flag state analysis tool to understand the erroneous behavior of TTT. Our tool produced a flag state diagram for the initial buggy version of TTT that contained too many nodes and edges to be understood..

Although we found it difficult to learn much about TTT from this initial diagram, we did observe many self-edges. We found these self-edges to be interesting, because they indicate that a task can potentially fire repeatedly on the same object. For example, we were able to use the initial diagram to determine that the `ProcessRequest` task can potentially fire multiple times. Since the `TTTServerSocket` object can only store a single request, this could potentially result in a race condition in which a second request clobbers the first request before the server can process the first request. To correct this bug, we modified the task specification to reset the `ReceiveRequest` flag after processing each request.

Figure 8 presents the flag state transition diagram after this correction. The nodes in this diagram represent the flag states of objects of the `TTTServerSocket` class and the edges represent the effects of task invocations on these objects. Double peripheries around a node indicate that new objects may be allocated with this flag state. A rectangular node indicates that objects in the flag state represented by the node will never reach a state in which no task can be invoked on it, and therefore such objects can never be garbage collected. We observed that the only node in this diagram with a double periphery is rectangular — this implies that `TTTServerSocket` objects can never be garbage collected.

We next observed that this new diagram contains self-edges for the `SendBoardDisplay`, `SendErrorMessage`, and `GameOver` tasks, indicating that these task may be executed repeatedly causing the server to display multiple copies of the same board, print error messages many times, and print the exit

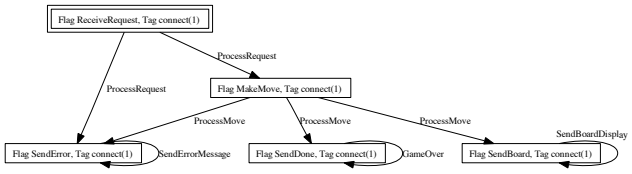

Fig. 8. Flag State Transition Diagram with `ReceiveRequest` reset

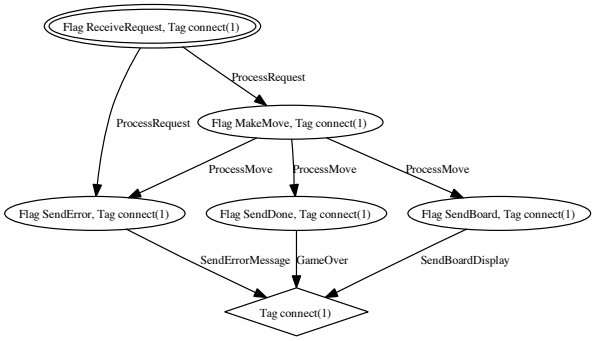

Fig. 9. Flag State Transition Diagram with Multiple Invocation Correction

message multiple times, respectively. Moreover, these possible repeated task invocations prevent these objects from being garbage collected. To correct these bugs, we modified the task specification to reset the `SendBoard`, `SendError`, and `SendDone` flags upon exiting the `SendBoardDisplay`, `SendErrorMessage`, and `GameOver` tasks, respectively.

Figure 9 presents the flag state transition diagram for the version that corrects these errors. Note that all of the nodes but one are elliptical, which indicates that the corresponding objects may eventually reach a garbage collectible state. The remaining diamond shaped node indicates that objects in this flag state will no longer be parameters of tasks and can be garbage collected if no references keep them alive. Finally, we observed that there are no paths from the `SendBoardDisplay` task or the `SendErrorMessage` task to the `ProcessRequest` task in the flag state transition diagram. This implies that the player can only make one move in the game. To correct this bug, we modified the task specification to set the `ReceiveRequest` flag upon exiting the `SendBoardDisplay` and `SendErrorMessage` tasks. Figure 10 presents the flag state transition diagram for the final version of TTT. Note that the `SendBoardDisplay` task and the `SendErrorMessage` task now return the object to the `Flag ReceiveRequest, Tag connect(1)` flag state where the `ProcessRequest` task is active.

B. Web Server

The web server benchmark contains features that are intended to closely resemble an e-commerce server. The web server maintains an inventory of merchandise and supports requests to perform commercial transactions on this inventory, including adding new items, selling items, and printing the inventory. We used the flag-state analysis tool to explore the behavior of this benchmark.

Figure 11 presents the flag state transition diagram for the

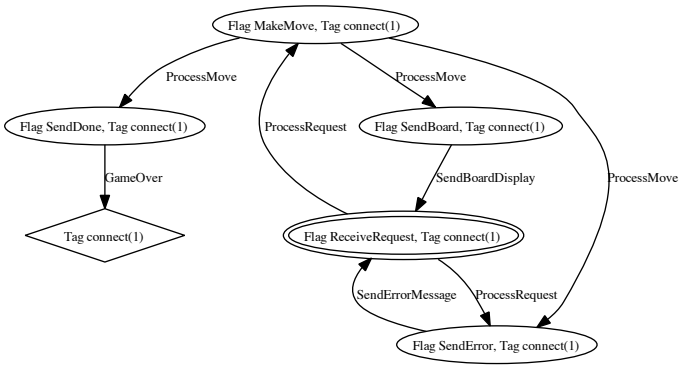

Fig. 10. Final Flag State Transition Diagram

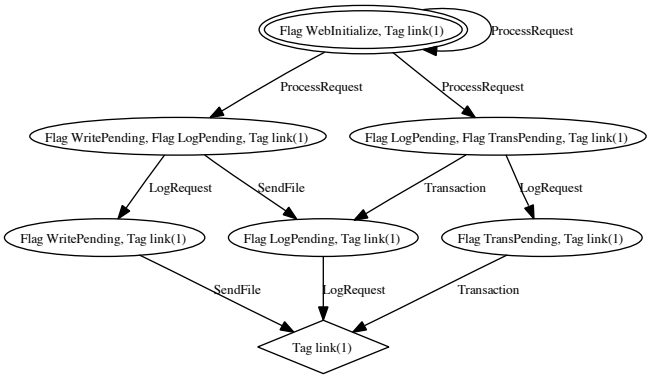

Fig. 11. Flag State Transition Diagram for the WebServerSocket class

`WebServerSocket` class. The nodes in this graph represent the flag states for the `WebServerSocket` class and the edges represent the task transitions. The absence of rectangular nodes in the graph indicates that a `WebServerSocket` object can reach a state in which it will be garbage collected unless a reference keeps it alive.

The diagram shows the two possible paths that a user request can take through the web server: a user may request the web server to serve a file or to perform a transaction on the inventory. The absence of a path from the `(Tag Link(1))` node to the start node `(Flag WebInitialize, Tag Link(1))` implies that the web server serves users on a single request basis. We also made an interesting observation: writing to the log is independent of serving the user request — they can be performed in either order.

Figure 12 shows the flag state transition diagrams for the `Inventory` and the `Logger` classes. These two diagrams show that instances of these classes are live for the lifetime of the web server. Inspection of the code reveals the reason that these objects are live for the entire execution of the web server — a single `Inventory` object is used to store the inventory of the e-commerce server and a single `Logger` object manages access to the log file.

Fig. 12. Flag State Transition Diagrams for the Inventory and Logger classes

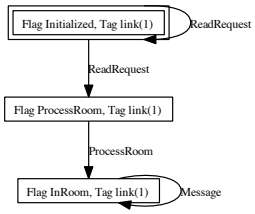

Fig. 13. Flag-State Transition Diagram for the ChatSocket Class

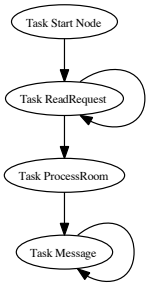

Fig. 14. Task Diagram for the ChatSocket Class

C. Chat Server

The multi-room chat server benchmark accepts incoming connections, asks the user to create a new room or select an existing room, and then allows the user to chat with the other users in that chat room. We explored the behavior of the chat server using our flag state analysis tool.

Figure 13 presents the flag-state transition diagram for the `ChatSocket` class. The presence of rectangular nodes indicate a limitation in the chat server — `ChatSocket` objects can never be garbage collected. We inspected the code and discovered that the issue is that the chat server does not contain functionality to allow the user to exit the chat room.

The flag state transition diagram for the `RoomObject` class contains a single node with a self-edge. The presence of the self-edge in that diagram indicates that this object can never be garbage collected. We inspected the code and discovered that the chat server uses a single instance of this class to maintain the list of chat rooms, and therefore, this is the desired behavior.

In many cases, a developer may wish to see a courser abstraction of an application's behavior. The task diagram is designed to further abstract the interaction patterns between tasks and objects. Figure 14 presents the task graph for the `ChatSocket` class. The nodes in this graph represent the tasks that act upon the `ChatSocket` object. From this diagram, we can see that after a new user connects to the chat server, the chat server reads the user's chat room request, processes this request, and then processes any messages that the user sends to the room.

D. Discussion

In general, the state transition diagrams helped us to quickly understand the structure of Bristlecone programs and to find and correct bugs in the task specifications. Based on our

experience, we believe that this tool will make writing correct task specifications even easier.

Our experience using the task state analysis to find bugs in TTT raises an important question: If Bristlecone is designed to tolerate software bugs, why did we use the static analysis to explore bug in a program? Note that there is an important distinction to be made between bugs in the code and bugs in the task specifications. Bristlecone is primarily designed to address bugs in the actual code. Bristlecone relies on correct task specifications to correctly compose the program: errors in the task specifications can result in the application exhibiting surprising behaviors.

However, our experience leads us to believe that writing correct task specifications is easier than writing correct code. In our experience, task specifications tend to be simple — they express how high-level operations in the software system interact. Errors in task specifications have been readily apparent within the first few executions of a Bristlecone program. We believe that this observation is a result of the high-level nature of task specifications and will hold across a wide range of Bristlecone programs. Furthermore, task specifications are amenable to static analysis, like the one presented in this behavior, that can help the developer understand all of the possible behaviors of the program.

While we selected a student-written program with task specification bugs because it was an interesting case study for our analysis, we believe that this benchmark represents an exception rather than the rule. We do not know of any task specification bugs in the several other Bristlecone programs written by others. It is important to remember that TTT was written by a student that did not have access to any Bristlecone tutorial, assistance from Bristlecone developers, or the static analysis developed in the paper.

VII. Related Work

We survey related work in testing, static analysis, exception mechanisms, fault tolerance, programming languages, and software architectures.

A. Program Understanding Tools

Daikon [16], [17] extracts likely algebraic invariants from information gathered during the program's execution. For example, Daikon can infer invariants such as "$y = 2x$". Our work differs in that the properties we infer focus on the interactions between objects and task and are guaranteed to hold for all executions.

Womble [27] and Chava [30] both use a static analysis to automatically extract object models for Java programs. Both tools use information from the class and field declarations; Womble also uses a set of heuristics to generate conjectures regarding associations between classes, field multiplicities, and mutability.

Unlike our tool, Womble and Chava do not support the concept of an object that changes state during the execution of the program. They instead statically group all instances of the same class into the same category of objects in the object model, ignoring any conceptual state changes that may occur.

Our previous work on role-based exploration of programs used dynamic analysis to automatically extract the role of objects and the interactions between these objects and the code [14]. Unlike our current tool, the previous tool used a dynamic analysis and, therefore, the validity of the extracted information depends on using an set of adequate test cases.

B. Approaches to Reliable Software

The standard approach to dealing with software errors is to work hard to prevent any inconsistencies from occurring in the first place. Approaches such as extensive testing [7], static analysis [19], [44], software model checking [11], error correction codes [41], and software isolation mechanisms [1] are all designed, in part, to eliminate as many potential data structure corruption errors as possible. We expect that Bristlecone will complement these other techniques: Bristlecone will enable software systems to recover from software errors that the other techniques do not catch. However, Bristlecone relies on these other techniques to ensure that the software system does not contain so many errors that the software system fails to perform any useful work.

Many programming languages, including Java, provide an exception handling mechanism [20]. This mechanism is intended to facilitate handling erroneous conditions. One issue with exceptions is that it is very difficult for developers to reason about which instructions are likely to throw exceptions. Java field accesses, array accesses, and numerical operations can all potentially throw exceptions — therefore, it turns out that a significant percentage of all Java instructions can potentially throw an exception. Moreover, it is difficult for developers to reason about how to resume the computation after an exception — the developer does not know how the computation will fail and the failure may leave the computation in an inconsistent state.

Software fault tolerance researchers have developed many techniques to address software failures. These techniques include recovery blocks [3], N-version programming [5], checkpointing [47], [35], [9], [46], and forward recovery [26]. These techniques typically have one of two shortcomings: they either require significant developer effort to create alternative implementations of the application or they are unable to execute past a deterministic failure.

Databases utilize transactions to ensure that the database is never left in a half-updated state by a partially completed sequences of operations [21]. Transactions ensure that either all the operations or none of the operations update the database. Researchers have developed software transactional memory to provide transactions to software systems as an alternate synchronization method [40], [2], [22], [43], [23].

Self-checking software is a general term that refers to software that verifies certain aspects of its own correct execution [45]. These aspects include the function of a process, the control sequence of a process, and the data of a process. If the software detects a failure, it may then take corrective action to recover.

There has recently been renewed interest in developing recovery mechanisms for hardware and software systems. The

Recovery-Oriented Computing project has explored integrating an undo operation into software systems [34], constructing systems out of a set of individually rebootable components [8], and developing redundant hardware systems. Failure oblivious computing is designed to address memory errors in C programs [37]. It detects erroneous memory operations and discards illegal write operations and manufactures values for invalid read operations. DieHard handles similar memory errors by using replication and randomization of the memory layout [6]. The randomization probabilistically ensures that illegal memory operations can only damage data structures in one of the replicants.

Researchers have developed a specification-based repair system that automatically generates repair algorithms from declarative consistency specifications [15] and imperative consistency checking code [29]. This technique enables software systems to recover from data structure consistency errors. The results from this research indicate that the generated repair algorithms can effectively repair inconsistent data structures in these software systems to enable the software systems to continue to operate successfully in cases where the original application would have failed.

Researchers have used meta-languages to decompose numerical computations into parallelizable tasks [36]. This technique is applicable to parallelizable numerical computations that compute the answers to many subproblems and then combine these answers to compute an overall answer. If one of the subcomputations fails, this approach simply ignores the failure. The developer uses random sampling to estimate how likely a failure is to yield unacceptable results. If a task fails, the system would use this estimate to determine how likely the computed result is to be acceptable. Bristlecone is designed to handle a broader class of software systems including servers, control systems, and office applications. Bristlecone is designed for software systems that may require stronger correctness guarantees; Bristlecone uses consistency checking and transactions to prevent software errors from corrupting critical data structures. Bristlecone uses the task specifications to reason which tasks are safe to invoke to continue the application's execution after a failure.

C. Related Languages

A key component of Bristlecone is decoupling unrelated conceptual operations and keeping track of data dependences between these operations. This part of the research is closely related to the dataflow computational model. Dataflow computation keeps track of data dependences between operations so that the operations can be parallelized [28]. Dataflow programs consist of a set of operations connected by queues. However, dataflow languages are not design to handle failures. Failures will either cause corrupt values to be placed in the queues, likely further propagating the error, or cause an operation to fail to place any value in the queue (possibly causing other operations to pair the wrong values together).

Tuple-space languages, such as Linda [18], also decouple computations to enable parallelization. These languages provide a set of primitives that the threads of execution use to communicate. These languages include primitives that add, read, and remove tuples of values from a global tuple-space. However, these language were not designed to address software errors. Software errors can permanently halt threads of execution in these languages causing the system to eventually fail. Automatically restarting these threads is unlikely to work as critical local state may have been lost. Furthermore, the communication primitives can be used in a very general fashion — automatically analyzing the communication patterns is generally difficult.

The orchestration language Orc [10] specifies how work flows between tasks. Orc is designed to decouple operations and expose parallelism. Note that if an operation fails, any work (and any corresponding data) flowing through the task may be lost. Since the goal of Orc is not failure recovery, it was not designed to contain mechanisms to recover data from failed tasks. Therefore, errors can cause critical information to disappear, eventually causing the software system to fail. Bristlecone uses flags to keep track of the conceptual states (or roles) that objects are in, enabling software systems to recover data from software errors and to continue to execute successfully.

Actors communicate through messages [25]. Actors were originally designed as a concurrent programming paradigm. Failures may cause actors to drop messages and corrupt or lose their state. Bristlecone's objects persist across task failures and can still be used by other tasks. Moreover, state corruption in actors can cause actors to permanently crash. Since Bristlecone's tasks are stateless, a previous failure of task do not affect future invocations of the task on different inputs.

Argus is a distributed programming language that organizes processes under guardians and isolates a process failure to the guardian under which it executes [31]. Inconsistency can cause the enclosing guardian to shut down. Guardians are stored in a persistent store and persist across failures of the underlying machine. Argus supports failure recover through an exception handling mechanism. This approach is complementary to Bristlecone: a developer can write exception handlers for anticipated failures and Bristlecone can be used to recover from unexpected failures.

Oz is a concurrent, functional language that organizes its computation as a set of tasks [42], [32]. Tasks are created and destroyed by the program. A task becomes reducible (executable) once its guard is satisfied by the constraint store. Task reducibility is monotonic — once a task is reducible it is always reducible. Task activation in Bristlecone is not monotonic, and allows a developer to temporarily disable a task if, for example, other tasks have placed an object into a state that is incompatible with the task or if the effect of task is no longer desirable. Non-monotonicity also allows developers to use different task to specify multiple implementations of the same functionality for redundancy. Moreover, since task creation is controlled by the program in Oz, it is more difficult to reason statically about task. For example, it appears difficult to automatically generate diagrams to show how Oz tasks and objects interact.

Concurrent Prolog is logic-based language that uses unifi-

cation to prove a goal [39], [38]. The proof corresponds to the execution of the program. Concurrent Prolog's guarded notation is similar to Bristlecone's flag expressions, but Concurrent Prolog's evaluation strategy starts from an end goal and reasons backwards. Concurrent Prolog programs may be able to recover from some failures by finding a different execution that reaches the same end goal. The downside of this approach is that if a failure prevents the program from completely achieving its end goal, the program will be unable to make partial progress. Bristlecone works forward from an initial flag setting for the startup object. A task is invoked if the heap contains objects with flags that satisfy the guards of the task's parameters. This evaluation strategy can make progress even if a failure prevents the system from completely achieving its goal.

Erlang has been used to implement robust systems using a software architecture containing a set of supervisors and a hierarchy of increasingly simple implementations of the same functionality [4]. The set of supervisors monitor the computation for errors. If an error is detected, the system falls back to a simpler implementation in the hierarchy. This approach has been used to develop reliable telephone switches. The two approaches are complementary — while the supervisor approach gives the developer complete control of the recovery process, the downside of this approach is that it requires the developer to manually develop multiple implementations of the same functionality. Bristlecone requires minimal develop effort — it can automatically perform recovery using only the task declarations. Furthermore, while a shared but minor fault could cause the entire Erlang implementation hierarchy to fail, in many cases Bristlecone may be able to execute around the fault and still provide nearly complete functionality.

VIII. Conclusion

Our experience shows that flag state transition diagrams and task diagrams were valuable abstractions for understanding the behavior of our benchmarks. We have implemented a static flag state analysis tool and a web-based, graphical user interface that helps developers explore the behavior of software applications. Our experience with several Bristlecone programs indicates that the tool can be useful for understanding and correcting software bugs and understanding the interaction between tasks and objects. Other potential applications include statically verifying properties of an application's execution, understanding the possible consequences of a failure, and providing a connection between an application's high-level design and the application's implementation.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, 1986.

[2] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High Performance Computer Architecture (HPCA-11)*, February 2005.

[3] T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 447–457, 1976.

[4] J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Swedish Institute of Computer Science, November 2003.

[5] A. Avizienis. The methodology of n-version programming, 1995.

[6] E. Berger and B. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates, 2002.

[8] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS-VIII*, pages 110–115, May 2001.

[9] K. M. Chandy and C. Ramamoorthy. Rollback and recovery strategies. *IEEE Transactions on Computers*, C-21(2):137–146, 1972.

[10] W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in Orc. In *Proceedings of the 2006 International Conference on Coordination Models and Languages (COORDINATION)*, 2006.

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

[12] B. Demsky, C. Cadar, D. Roy, and M. C. Rinard. Efficient specification-assisted error localization. In *Proceedings of the Second International Workshop on Dynamic Analysis*, 2004.

[13] B. Demsky and A. Dash. Bristlecone: A language for robust software systems. `http://newport.eecs.uci.edu/~bdemsky/bristlecone.pdf`, November 2006.

[14] B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[15] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 2005 International Conference on Software Engineering*, May 2005.

[16] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *International Conference on Software Engineering*, pages 449–458, 2000.

[17] M. D. Ernst, Y. Kataoka, W. G. Griswold, and D. Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington, November 1999.

[18] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[19] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 1996.

[20] J. B. Goodenough. Structured exception handling. In *POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1975.

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

[22] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 Conference on Programming Language Design and Implementation*, June 2006.

[23] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 14–25, New York, NY, USA, 2006. ACM Press.

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

[25] C. Hewitt and H. G. Baker. Actors and continuous functionals. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[26] K. Huang, J. Wu, and E. B. Fernandez. A generalized forward recovery checkpointing scheme. In *Proceedings of the 1998 Annual IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, April 1998.

[27] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *International Conference on Software Engineering*, pages 194–202, 1999.

[28] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1), 2004.

[29] S. Khurshid, I. García, and Y. L. Suen. Repairing structurally complex data. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN)*, August 2005.

[30] J. Korn, Y.-F. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of Java applets. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 314–325, October 1999.

[31] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl. Argus reference manual. Technical Report MIT-LCS-TR-400, Massachusetts Institute of Technology, November 1987.

[32] M. Mehl. *The Oz Virtual Machine - Records, Transients, and Deep Guards*. PhD thesis, Technische Fakultät der Universität des Saarlandes, 1999.

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

[34] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kcman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, March 15, 2002.

[35] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[36] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th ACM International Conference on Supercomputing*, 2006.

[37] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, December 2004.

[38] E. Shapiro. Concurrent Prolog: A progress report. *Computer*, 19(8):44–58, 1986.

[39] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[40] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, August 1995.

[41] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey. Software-implemented EDAC protection against SEUs. *IEEE Transactions on Reliability*, 49(3):273–284, September 2000.

[42] G. Smolka. The Oz programming model. In *Proceedings of the European Workshop on Logics in Artificial Intelligence*, page 251, London, UK, 1996. Springer-Verlag.

[43] M. F. Spear, V. J. Marathe, W. N. Schereer, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the Twentieth International Symposium on Distributed Computing*, 2006.

[44] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, 2006.

[45] S. S. Yau and R. C. Cheung. Design of self-checking software. In *Proceedings of the International Conference on Reliable Software*, pages 450–455, 1975.

[46] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[47] Y. Zhang, D. Wong, and W. Zheng. User-level checkpoint and recovery for LAM/MPI. *ACM SIGOPS Operating Systems Review*, 39(3):72–81, 2005.