Bristlecone: Language Support for Robust Software Applications

Brian Demsky and Sivaji R. Sundaramurthy

Abstract—We present Bristlecone, a programming language for robust software systems. Bristlecone applications have two components: a high-level organization specification that describes how the application's conceptual operations interact and a low-level operational specification that describes the sequence of instructions that comprise an individual conceptual operation. Bristlecone uses the high-level organization specification to recover the software system from an error to a consistent state and to reason how to safely continue the software system's execution after the error. We have implemented a compiler and runtime for Bristlecone. We have evaluated this implementation on three benchmark applications: a Web crawler, a Web server, and a multiroom chat server. We developed both a Bristlecone version and a Java version of each benchmark application. We used injected failures to evaluate the robustness of each version of the application. We found that the Bristlecone versions of the benchmark applications more successfully survived the injected failures. The Bristlecone compiler contains a static analysis that operates on the organization specification to generate a set of diagrams that graphically present the task interactions in the application. We have used the analysis to help understand the high-level structure of three Bristlecone applications: a game server, a Web server, and a chat server.

Index Terms—Software robustness.

1 Introduction

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

The Lucent 5ESS telephone switch, the Ericsson AXD301 ATM switch, and the IBM MVS operating system are examples of critical systems that use recovery routines to automatically recover from software failures [24], [33]. The software in these systems contains a set of manually coded recovery procedures that detect errors and then take actions to automatically recover from the errors. The reported results indicate that the recover routines can provide an order of magnitude increase in the reliability of these systems [22]. This additional reliability comes at a significant additional development cost—the recovery routines for the Lucent 5ESS telephone switch constitute more than 50 percent of the switch's software [7]. As a result of these high costs, recovery procedures have been primarily relegated to the domain of critical infrastructure software that can justify the cost. A wide range of other applications, including desktop applications such as Web browsers, office applications, games, servers, and control systems, could potentially benefit from lower cost automated recovery. The goal of Bristlecone is to provide a lower cost approach to software recovery that will enable a larger class of applications to benefit from this technique.

1.1 Bristlecone Language

The key inspiration for this research is the observation that many software errors propagate through software systems to cause further damage either through data structure corruption or control-flow-induced coupling between *conceptual operations* (i.e., high-level application operations). We can view software systems as a composition of many conceptual operations—in practice, the correct execution of any operation is likely to be independent of most other operations. However, many traditional programming languages force developers to linearize the conceptual operations in a software system. This linearization tightly couples these conceptual operations: If one conceptual operation fails, it becomes unclear how to safely execute any future conceptual operations.

We have developed Bristlecone, a programming language for robust software systems, to address the error propagation problem. Bristlecone extends a core Java-like, object-oriented language[1] with a set of task-based extensions. The basic idea is to construct software systems as a set of decoupled tasks with each task encapsulating one of the conceptual operations that comprise the application. A set of specifications describes how these decoupled tasks interact. The runtime uses these task specifications to determine when to invoke tasks. The runtime checks for data structure consistency violations and monitors for

• *The authors are with the Department of Electrical Engineering and Computer Science, The Henry Samueli School of Engineering, University of California, Irvine, 3213 Engineering III, Irvine, CA 92697-2625. E-mail: bdemsky@uci.edu, sivaji.raju@gmail.com.*

Manuscript received 6 Sept. 2008; revised 17 Feb. 2009; accepted 30 July 2009; published online 4 Feb. 2010.
Recommended for acceptance by S.C. Cheung.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-09-0281.
Digital Object Identifier no. 10.1109/TSE.2010.27.

1. Bristlecone excludes several Java features such as threading, globals variables, and reflection.

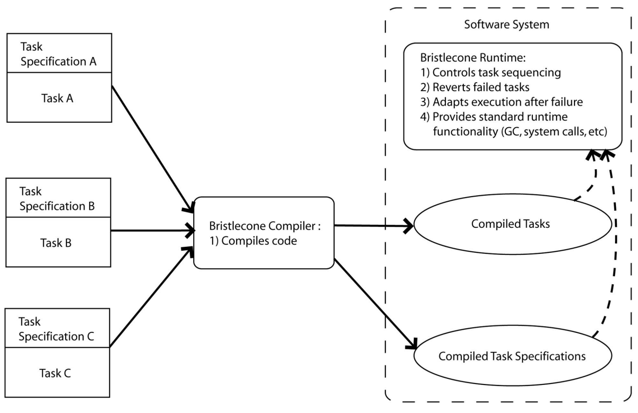

Fig. 1. Overview of the Bristlecone system.

illegal operations (such as illegal memory accesses or arithmetic errors) to detect software errors. When the runtime detects an error in the execution, the runtime rolls back the data structures to their state at the beginning of the task's execution and then uses the task specifications to adapt the execution of the software system to avoid reexecuting the same error and make forward progress.

Fig. 1 gives an overview of the components in the Bristlecone system. Bristlecone applications are composed of a set of tasks. Tasks are represented in Fig. 1 as rectangles. A set of task specifications describes how the runtime should orchestrate the execution of the individual tasks that comprise an application. Moreover, if a task fails, the runtime uses the task specifications to reason how to adapt the future execution of the software system so that the execution does not depend on the failed task.

Bristlecone contains the following components (represented by rounded boxes in the figure):

- **Bristlecone compiler:** The Bristlecone compiler compiles the tasks and task specifications into C code. We chose to target C because it frees us from implementing the low-level details of targeting machine code such as instruction selection while still retaining most of the flexibility. This flexibility has proven useful to efficiently support rollback of data structures. Our implementation then uses the gcc C compiler to generate executables. The ellipse labeled `Compiled Tasks` represents the compiled tasks.
- **Runtime:** The runtime uses the compiled code and compiled specifications (represented by the ellipses in the figure) to execute the software system. If the runtime detects a software error, it aborts the task's enclosing transaction to recover consistent data structures. Finally, it uses the task specifications to determine when to execute the tasks and how to recover from errors.

1.2 Scope

Bristlecone is not suitable for all software systems. Certain computations, such as some scientific simulations, are inherently tightly coupled. While Bristlecone may detect errors in such software systems, it is unlikely to enable these systems to recover in any meaningful way. For other computations, it may be desirable for a software system to shut down rather than deviate from a specific designed behavior or produce a partial result.

Bristlecone is designed for software systems that place a premium on continued execution and that can tolerate some degradation from a specific designed behavior. For example, we expect that Bristlecone will be useful for financial server software, e-commerce systems, office applications, Web browsers, online game servers, sensor networks, and control systems for physical phenomena. For applications like finance, Bristlecone can be used to develop software systems that only process error-free transactions and back out all changes that corrupt data structures, while still ensuring that cosmetic errors do not cause potentially expensive downtime. Ultimately, the software developer must decide whether using this approach is reasonable for a given software system.

This decision could depend on the environment in which a system is deployed. For example, in systems with redundant backup systems, we expect that developers would design the primary system to fail-fast and the backup system to be robust in the presence of errors.

Bristlecone can only generate meaningful recovery for computations that contain multiple independent subcomputations. Examples of ideal applications include applications that react to many independent inputs, interactive office applications that react to many user requests, servers that service independent requests, and games that contain multiple independent actors. Bristlecone is unlikely to generate meaningful recovery for tightly coupled numerical computations in which the output depends on every operation.

Bristlecone requires developers to identify the high-level conceptual operations that comprise an application and then structure the application as a set of tasks that implement these operations. In some cases, it may be difficult to decompose legacy applications or applications that rely heavily on existing library code into collections of tasks. It is possible that expressing certain types of computations as tasks could be cumbersome—for example, code that contains complex communications interactions may require large numbers of tasks to implement.

1.3 Contributions

This paper makes the following contributions:

- **Bristlecone language:** It presents a set of extensions to a core Java-like object-oriented language that exposes both the conceptual operations and the ordering and data dependencies between these conceptual operations to the compiler and runtime system.
- **Recovery strategy:** It presents a strategy for repairing the damage caused by a software error and adapting the software system's execution in response to the error to enable it to safely continue execution.
- **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 abstracted states of an object's flags and tags and the edges represent the changes the tasks induce on the object's flags and tags. This graph is intended to help developers visualize the interactions 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.

- **Experience:** It presents our experience using Bristlecone to develop three robust software systems: a Web crawler, a Web server, and a multiroom chat server. For each benchmark, we developed both a Bristlecone version and a Java version. We designed the Java versions to be resilient: They use threads to tolerate failures. Our experience indicates that the Bristlecone versions are able to successfully recover from significantly more of the injected failures.

  It also presents our experience using the static analysis to help understand three applications: a tic-tac-toe server, a Web server, and a multiroom chat server. For each benchmark application, we examined both the flag state transition diagram and the task diagram to understand the interactions of task in the application.

The remainder of the paper is structured as follows: Section 2 presents an example that illustrates our approach. Section 3 presents the Bristlecone languages. Section 4 discusses the runtime system. Section 5 presents a static task specification analysis. Section 6 presents our experience using Bristlecone to develop several robust software applications. Section 7 describes our experience using the task specification analysis to explore several Bristlecone applications. Section 8 compares Bristlecone with related work; we conclude in Section 9.

2 Example

We next present a Web server example that illustrates the operation of Bristlecone. The example uses an event-driven architecture [35].

2.1 Objects

The Web server uses `WebRequest` objects to track the state associated with an individual connection. Fig. 2 gives part of the `WebRequest` class definition. Bristlecone classes are similar to Java—Bristlecone classes support standard object-oriented constructs including inheritance, methods, and virtual dispatch. The Web server example uses instances of the `WebRequest` class to manage connections to the Web server.

As the example Web server executes, the conceptual state or role of objects in the computation evolves. This evolution changes the way that the software system uses the object and can change the functionality that the object supports. Bristlecone uses *flags* to track the conceptual state of an object. The developer declares a flag in a class with the `flag` keyword followed by the flag's name. The runtime then uses the conceptual state of an object as indicated by the object's flags to determine which *tasks* to invoke on the given object. When a task exits, it can change the values of the flags of its parameter objects.

```
class WebRequest {
  /* This flag indicates that the WebRequest
     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. 2. WebRequest class declaration.

The `WebRequest` class definition declares three flags: the `initialized` flag, which indicates whether the connection is in the 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. For example, the example Web server must ensure that tasks operate on a `Socket` object and a `WebRequest` object from the same connection. To address this issue, Bristlecone provides a tag construct which the developer can use to group objects together. New tag instances 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 in a task specification to ensure that the runtime invokes a task on objects bound to the same tag.

Bristlecone adds a modified `new` statement that specifies the initial flag settings and tag bindings for a newly allocated object. These take effect when the task exits. Bristlecone also contains a `taskexit` statement that specifies how the task changes the state of the flags or tag bindings of its parameter objects at that task exit point. The example uses the `connection` tag to group a `WebRequest` object with the corresponding `Socket` object that provides the TCP connection for that Web request.

2.2 Tasks

We next discuss Bristlecone tasks. Each task declaration consists of the keyword `task`, the task's name, the task's parameters, and the body of the task. Each task parameter

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

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

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

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

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

Fig. 3. Flag specifications for tasks.

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 the tag instances that the parameter's guard expression specifies.

The key difference between tasks and methods is that the runtime manages task invocation—the runtime can invoke a task when all of its parameter objects are available in the heap. In other words, Bristlecone's tasks are implicitly invoked [18] by the runtime when a set of parameter objects transition into a state that satisfies the task's guards. Note that while the runtime controls task invocation, tasks can call methods. Bristlecone methods have identical properties as Java methods.

Fig. 3 gives the task declarations for the Web server example. We indicate the omission of the Java-like imperative code inside the task declarations with ellipses. The first task declaration declares that the task named `startup` takes a `StartupObject` object as a parameter and points the parameter variable `start` at this object. The declaration also

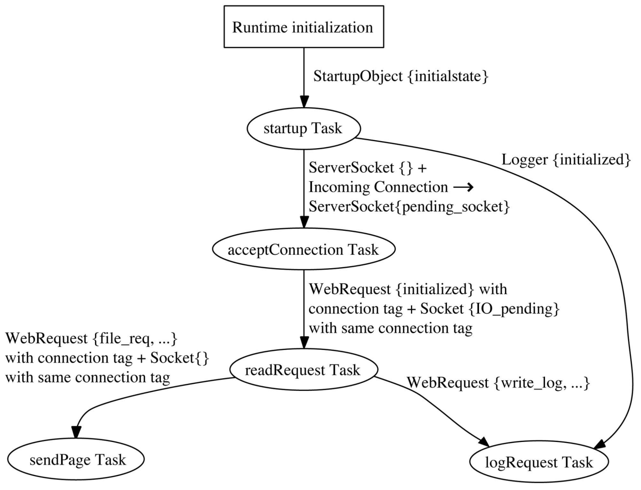
Runtime initialization
StartupObject {initialstate}
startup Task
Logger {initialized}
ServerSocket {} + Incoming Connection ⟶ ServerSocket{pending_socket}
acceptConnection Task
WebRequest {initialized} with connection tag + Socket {IO_pending} with same connection tag
WebRequest {file_req, ...} with connection tag + Socket{} with same connection tag
readRequest Task
WebRequest {write_log, ...}
sendPage Task
logRequest Task

Fig. 4. Task diagram for the Web server.

contains a guard that states that the `StartupObject` object must have its `initialstate` flag set before the runtime can invoke this task. The runtime invokes the task when there exist parameter objects in the heap that satisfy the parameters' guard expressions. Before exiting, the `taskexit` statement in the `startup` task resets the `initialstate` flag in the `StartupObject` to false to prevent the runtime from repeatedly invoking the `startup` task.

Task declarations can contain constraints on tag bindings to ensure that the parameter objects are related. A tag binding constraint for a parameter contains the keyword `with` followed by the type of the tag and the tag variable. For example, the task declaration `task readRequest (WebRequest w in initialized with connection t, Socket s in IO_Pending with connection t)` ensures that the runtime only invokes the `readRequest` task on a set of parameter objects in which the first parameter object is bound to an instance of a `connection` tag and the second parameter object is bound to the same `connection` tag instance. When the task executes, the tag variable `t` is bound to that `connection` tag instance.

If multiple objects satisfy a task, the runtime nondeterministically selects a given parameter object binding to execute first. We expect that developers will use this functionality to process independent computations. Similarly, if a given object satisfies multiple tasks, the runtime nondeterministically selects a given task to execute first. We expect that developers will use this functionality when operations can be safely reordered on a given object.

2.3 Error-Free Execution

Fig. 4 gives a diagram of the dependencies between tasks in the Web server example. The ellipses in the diagram represent tasks and the edges represent the control and data dependencies between the tasks. The rectangle labeled `Runtime initialization` represents the initialization performed by the Bristlecone runtime. From this diagram, we can see that the Web server performs the following operations in an error-free execution (although not necessarily in this order):

1. **Startup:** When a Bristlecone program is executed, the Bristlecone runtime creates a `StartupObject`

object and then sets its `initialstate` flag to true. Setting this flag causes the runtime to invoke the `startup` task in our example.

When the runtime invokes the `startup` task, the `startup` task creates a `ServerSocket` object to accept incoming connections to the Web server. Next, it creates a `Logger` object to manage logging Web page requests and sets its `initialized` flag to indicate that the object is ready to provide logging functionality. Finally, it resets the `StartupObject` object's `initialstate` flag to false to prevent the runtime from repeatedly invoking the `startup` task. Note that due to the transactional semantics of tasks, changes to an object's flags take effect when the task ends.

2. **Accepting an incoming connection:** At some point, the Web server will receive an incoming connection request from a Web browser. This causes the runtime to set the `ServerSocket` object's `pending_socket` flag to true, which in turn causes the runtime to invoke the `acceptConnection` task with this `ServerSocket` object as its parameter. The `acceptConnection` task creates a `WebRequest` object to store the connections state and calls the accept method on the `ServerSocket` to create a `Socket` object to manage communication with the Web browser. Note that the `acceptConnection` task creates a new `connection` tag instance to group the `Socket` object and `WebRequest` object together by binding this tag instance to the `WebRequest` object and then passing this tag instance into the `accept` method to bind the newly created `Socket` object.
3. **Reading a request:** After a 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,[2] which, in turn, causes the runtime to invoke the `readRequest` task. The `readRequest` task checks whether the server has received the complete request.[3] If it 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 object.
4. **Sending the page:** The runtime invokes the `sendPage` task when the `WebRequest` object's `request_processed` flag is set to true. 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 `file_req` flag to false to prevent repeated invocations of the `sendPage` task.
5. **Logging the request:** The runtime invokes the `logRequest` task when both the `WebRequest` object's `write_log` flag is set to true and the `Logger` object's `initialized` flag is set to true. 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.

2. The `IO_pending` flag is declared with the `external` keyword to indicate that the runtime manages setting and clearing this flag. The current runtime implementation of Bristlecone is single-threaded, and therefore, uses nonblocking I/O. Future runtime implementations will support multiple concurrent tasks and (transactional) blocking I/O [23].

3. Note that it is possible for a client browser to split a long request across multiple packets, and therefore, it may be necessary to invoke the `readRequest` task multiple times to receive a single request.

2.4 Error Handling

The Bristlecone runtime uses task specifications to automatically recover from errors. For example, suppose that the `logRequest` task fails while updating the `Logger` object. If the Web server were written in a traditional programming language, it could be difficult to recover from such a failure. While some traditional languages provide exceptional handling mechanisms, using them effectively is challenging—the developer must both identify which failures are likely to occur and reason about how to recover from those failures. Alternatively, the program could simply ignore the failure. Unfortunately, if the Web server were to simply ignore the failure, it could easily leave the `Logger` object in an inconsistent state, possibly eventually causing a catastrophic failure.

To address this issue, Bristlecone tasks have transactional semantics—upon failure, the Bristlecone runtime aborts the enclosing transaction to return the affected objects, including the `Logger` object, to consistent states. The runtime then records that the `logRequest` task failed when invoked on the specific combination of `WebRequest` and `Logger` objects. The runtime uses this record to avoid reexecuting the same specific failure. At this point, the runtime has returned the Web server to a known consistent state and must now determine how to safely continue the Web server's execution.

The traditional problem with using transactions to recover from deterministic software faults is that after aborting a transaction, the software system cannot make forward progress—retrying the same transaction will cause the system to repeat the same failure. Bristlecone solves this problem by using the task specifications to determine which other tasks are safe to execute after the error. Although the software fault prevents the system from logging this request since the `file_req` flag is set to true, the task specification for the `sendPage` task allows the runtime to invoke the `sendPage` task. Therefore, the runtime can still safely serve the Web page request.

The end result is that the software system is able to safely continue to execute even in the presence of software errors. Bristlecone is able to successfully isolate the effects of the error to a minimal part of the Web server's execution—only a single task is aborted and the abort is logged. Without Bristlecone, the Web server could potentially leave the `Logger` object in an inconsistent state, possibly causing the Web server to fail to log future requests. If the Web server written in a conventional language was designed to log a request before serving a request, corruption of the log data structure could even cause the server to stop serving requests.

2.5 Static Specification Analysis

We next discuss the operation of a static analysis of the task specifications using the Web server example. The static analysis generates diagrams that can help developers

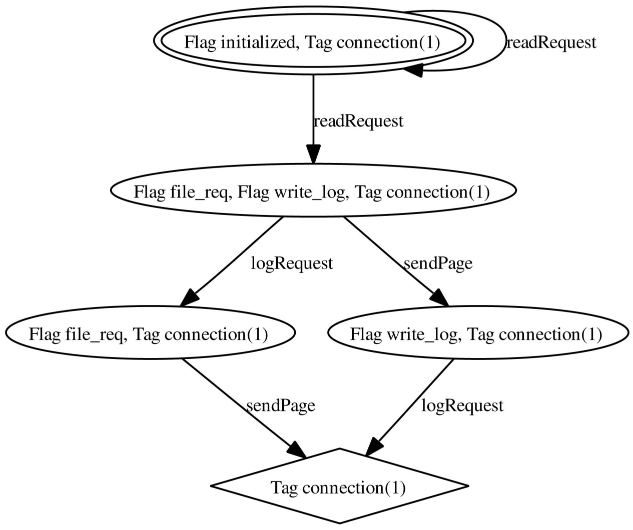
Flag initialized, Tag connection(1)
readRequest
readRequest
Flag file_req, Flag write_log, Tag connection(1)
logRequest
sendPage
Flag file_req, Tag connection(1)
Flag write_log, Tag connection(1)
sendPage
logRequest
Tag connection(1)

Fig. 5. Flag state transition diagram for the `WebConnection` class.

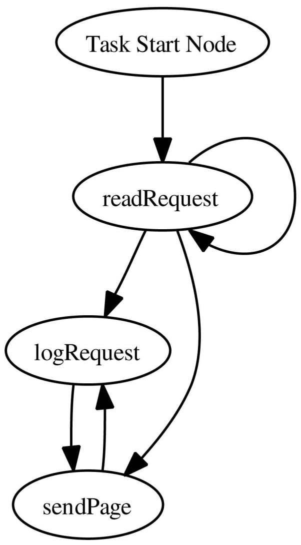

Fig. 6. Task diagram for the `WebConnection` class.

better understand the possible behaviors of Bristlecone applications.

2.5.1 Flag State Transition Diagrams

Fig. 5 presents a flag state transition diagram for the Web server example. Flag state transition diagrams capture the dependencies between tasks. The nodes in this diagram represent the flag states of objects and the edges represent transitions between these flag states that the tasks perform. Each 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 as `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 `Flag file_req, Flag write_log, Tag connection(1)` state. 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, 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.

2.5.2 Task Diagrams

Fig. 6 presents 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.

2.5.3 Understanding Consequences of Failures

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 Fig. 5 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 aborting the first task, thereby returning to the fork in the graph and then choosing an alternate path at that fork.

3 Language Design

The Bristlecone language includes a task specification language that describes how to orchestrate task execution. Bristlecone introduces object flags to store the conceptual state of the object. 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 with syntax similar to Java. Fig. 7 presents the grammar for Bristlecone's task extensions to Java. We omit the Java-like imperative component of Bristlecone from the grammar to save space.

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 handle asynchronous

$$\begin{array}{rcl}
\mathit{flagdecl} & := & \texttt{flag}\ \mathit{flagname};\ \mid \texttt{external flag}\ \mathit{flagname}; \\
\mathit{tagdecl} & := & \mathit{tagtype\ tagname}; \\
\mathit{taskdecl} & := & \texttt{task}\ \mathit{name}(\mathit{taskparamlist}) \\
\mathit{taskparamlist} & := & \mathit{taskparamlist}, \mathit{taskparam} \mid \mathit{taskparam} \\
\mathit{taskparam} & := & \mathit{type\ name}\ \texttt{in}\ \mathit{flagexp} \mid \mathit{type\ name}\ \texttt{in}\ \mathit{flagexp} \\
 & & \texttt{with}\ \mathit{tagexp} \\
\mathit{flagexp} & := & \mathit{flagexp}\ \texttt{and}\ \mathit{flagexp} \mid \mathit{flagexp}\ \texttt{or}\ \mathit{flagexp} \mid \\
 & & !\mathit{flagexp} \mid (\mathit{flagexp}) \mid \mathit{flagname} \mid \texttt{true} \mid \\
 & & \texttt{false} \\
\mathit{tagexp} & := & \mathit{tagexp}\ \texttt{and}\ \mathit{tagtype\ tagname} \mid \mathit{tagtype} \\
 & & \mathit{tagname} \\
\mathit{statements} & := & \ldots \mid \texttt{taskexit}(\mathit{flagactionlist}) \mid \\
 & & \texttt{assert}(\mathit{expression}) \mid \\
 & & \texttt{tag}\ \mathit{tagname}\ =\ \texttt{new tag}(\mathit{tagtype}) \mid \\
 & & \texttt{new}\ \mathit{name}(\mathit{params})(\mathit{flagortagactions}) \\
\mathit{flagactionlist} & := & \mathit{flagactionlist};\ \mathit{name}\ :\ \mathit{flagortagactions} \mid \\
 & & \mathit{name}\ :\ \mathit{flagortagactions} \\
\mathit{params} & := & \ldots \mid \texttt{tag}\ \mathit{tagname} \\
\mathit{flagortagactions} & := & \mathit{flagortagactions}, \mathit{flagortagaction} \mid \\
 & & \mathit{flagortagaction} \\
\mathit{flagortagaction} & := & \mathit{flagaction} \mid \mathit{tagaction} \\
\mathit{flagaction} & := & \mathit{flagname} := \mathit{boolliteral} \\
\mathit{tagaction} & := & \texttt{add}\ \mathit{tagname} \mid \texttt{clear}\ \mathit{tagname}
\end{array}$$

Fig. 7. Task grammar.

$$\begin{array}{rcl}
V & \in & Values = Objects + Primitives \\
m & \in & Memory = Objects \times Field \rightarrow Values \\
L & \in & TagVariables = Identifier \rightarrow TagInstance \\
\mathcal{F} & \in & FlagStatus = Objects \times Flag \rightarrow Boolean \\
\mathcal{T} & \in & TagStatus = Objects \times TagInstances \\
\tau & \in & InstanceType = TagInstances \rightarrow TagTypes \\
\mathcal{B} & \in & ParameterBinding = Identifier \rightarrow Objects \\
\mathcal{I} & \in & TaskInvocation = Task \times ParameterBinding
\end{array}$$

$$\begin{array}{rcl}
\mathcal{G} & \in & GlobalState = Memory \times FlagStatus \times TagStatus \times \\
 & & InstanceType \times TaskInvocation_{failed} \\
\mathcal{M} & \in & TaskState = Memory \times FlagStatus \times TagStatus \times \\
 & & InstanceType \times TagVariables \times ExecutionState \\
\mathcal{C} & \in & Configuration = GlobalState \times TaskInvocation_{current} \times \\
 & & TaskState
\end{array}$$

Fig. 8. Domain definitions.

events such as communication over the Internet or mouse clicks. External flags are intended to be declared in library code with the corresponding runtime component setting and clearing the external flag.

Bristlecone contains an `assert` statement that can be used to specify correctness properties that must hold. The goal of assert statements is to provide a mechanism to detect higher level errors that do not cause low-level exceptions. The compiled application uses the assert statements to detect errors at runtime—if it detects an error, the runtime system will invoke the recovery algorithm. These assertion statements can be used with data structure consistency checking tools [16], JML assertions [29], or design by contract methodologies [32]. In many cases, the assertions can be generated automatically using dynamic invariant detection tools [17].

3.1 Design Rationale

An obvious alternative to Bristlecone's flag construct is a workflow diagram similar to our flag state transition diagrams. The primary advantage of Bristlecone's formulation is that the developer can create several sets of flags that characterize orthogonal aspects of an object's abstract state. The developer then includes only the flags in a task specification for the components of an object's abstract state that concerns that task. If a task fails, this approach enables tasks that concern orthogonal aspects of the object's state to continue to execute. For example, if a Web request exposes an error in the code that serves the Web request, the logging task could still execute. Bristlecone's constructs give the developer much freedom—in some cases, a task may depend on multiple aspects of an object's state and the corresponding parameter guards can specify the flags for each relevant aspect.

While it is possible to express the same computation in the form of a diagram, the diagram must explicitly enumerate all legal sequences of task invocations. For example, if there are three independent tasks to be executed on a given object, the diagram approach requires the developer to enumerate all six possible sequences of these three tasks. In cases in which the diagram representation is more natural, it is straightforward to build automated tools that translate task diagrams into the corresponding task specifications.

3.2 Serial Semantics

This section develops operational semantics for the serial execution of Bristlecone programs.

3.2.1 Domains

Fig. 8 introduces the domains used by the operational semantics. This section develops an operational semantics that transforms machine configurations $\mathcal{C}$. Each machine configuration $\mathcal{C}$ consists of the global committed state $\mathcal{G}$, the executing task invocation $\mathcal{I}$, and the executing task's state $\mathcal{M}$. The executing task operates on the task state component $\mathcal{M}$ of the machine configuration. When this task commits, the flag, tag binding, and heap changes the task made to the task state $\mathcal{M}$ are committed to the global committed state $\mathcal{G}$.

A tag variable mapping L maps tag variable identifiers to tag instances. A flag status mapping $\mathcal{F}$ maps object flags to a Boolean that indicates whether the flag is set. A tag status mapping $\mathcal{T}$ maps objects to the set of tag instances that are bound to the object. A tag type mapping τ maps tag instances to their tag types. A binding mapping $\mathcal{B}$ maps parameter identifiers to the parameter objects that they reference. A task invocation $\mathcal{I}$ is composed of a task and a set of bindings for the task's parameter objects.

3.2.2 Helper Functions

We next define several helper functions that will simplify the presentation of Bristlecone's operational semantics. The $setflag$ helper function, given below, is used to update the FlagStatus $\mathcal{F}$ to reflect an operation that sets or clears the flag f:

$$\begin{aligned} setflag(o, f, bool, \mathcal{F}) = \{ & \langle o', f', b \rangle \in \mathcal{F} \mid o \neq o' \mid\mid f \neq f' \} \\ & \cup \{ \langle o, f, bool \rangle \}. \end{aligned}$$

The pair of helper functions *settag* and *cleartag* are used to update the TagStatus $\mathcal{T}$ to reflect operations that either add or remove, respectively, a binding between a tag instance and an object. These functions follow below:

$$settag(o,t,\mathcal{T}) = \mathcal{T} \cup \{\langle o,t\rangle\},$$
$$cleartag(o,t,\mathcal{T}) = \{\langle o',t'\rangle \in \mathcal{T} \mid o \neq o' \;||\; t \neq t'\}.$$

The *satisfies* helper function, given below, is used to determine whether an object's flags satisfied the guard expression *flagexp*:

$$\begin{aligned}
&satisfies(o,f,\mathcal{F}) = \mathcal{F}(o,f),\\
&satisfies(o,!flagexp,\mathcal{F})\\
&\quad = \neg satisfies(o,flagexp,\mathcal{F}),\\
&satisfies(o,flagexp_1 \texttt{ and } flagexp_2,\mathcal{F})\\
&\quad = satisfies(o,flagexp_1,\mathcal{F}) \wedge satisfies(o,flagexp_2,\mathcal{F}),\\
&satisfies(o,flagexp_1 \texttt{ or } flagexp_2,\mathcal{F})\\
&\quad = satisfies(o,flagexp_1,\mathcal{F}) \vee satisfies(o,flagexp_2,\mathcal{F}).
\end{aligned}$$

3.2.3 Task Invocation

We next define the task invocation transition. Without loss of generality, we express the task declaration in the following form:

$$\begin{aligned}
&\texttt{task } name(p_1 \texttt{ in } flagexp_1 \texttt{ with } tagtype_{11}\ tagname_{11},\ldots,\\
&\quad tagtype_{1m_1}\ tagname_{1m_1},\ldots,p_n \texttt{ in } flagexp_n \texttt{ with } tagtype_{n1}\\
&\quad tagname_{n1},\ldots,tagtype_{nm_n}\ tagname_{nm_n}).
\end{aligned}$$

To invoke a task on a set of parameter objects, the implementation must ensure that this task has not already failed on the same set of parameter objects $I \notin \mathcal{I}_{failed}$, the parameter objects satisfy the flag guard expressions B_{flags}, the parameter objects satisfy the tag guard expressions B_{tags}, and the implementation is not currently executing a task. The implementation has the freedom to invoke any task that satisfies these conditions. We define the transition for the task invocation operation below. Note that the object variables in $\mathcal{B}$ quantify over all objects in the heap and the tag variables in L quantify over all tag instances:

$$\frac{I \notin \mathcal{I}_{failed}, B_{flags}, B_{tags}}{\langle\langle m,\mathcal{F},\mathcal{T},\tau,\mathcal{I}_{failed}\rangle,\emptyset,\emptyset\rangle \Rightarrow \langle\langle m,\mathcal{F},\mathcal{T},\tau,\mathcal{I}_{failed}\rangle,I,\mathcal{M}\rangle},$$

$$\begin{aligned}
\texttt{where}\quad I &= \langle name,\mathcal{B}\rangle,\\
\mathcal{B} &= \{\langle p_1,o_1\rangle,\ldots,\langle p_n,o_n\rangle\},\\
B_{flags} &= 1 \le i \le n.\ satisfies(o_i,flagexp_i,\mathcal{F}),\\
B_{tags} &= 1 \le i \le n.\ 1 \le j \le m_i.\langle t_{ij},tagtype_{ij}\rangle\\
&\quad \in \tau, \langle o_i,t_{ij}\rangle \in \mathcal{T},\\
L &= \{\langle tagname_{11},t_{11}\rangle,\ldots,\langle tagname_{1m_1},t_{1m_1}\rangle,\ldots,\\
&\qquad \langle tagname_{n1},t_{n1}\rangle,\ldots,\langle tagname_{nm_n},t_{nm_n}\rangle\},\\
\mathcal{M} &= \langle m,\mathcal{F},\mathcal{T},\tau,L,\mathcal{S}_{fresh}\rangle,
\end{aligned}$$

where $\mathcal{S}_{fresh}$ is a fresh execution state for the task to run in.

Note that $\mathcal{S}_{fresh}$ represents a new task execution state (i.e., stack, processor registers, etc.).

3.2.4 Task Commit

We next define the task commit transition. Without loss of generality, we express the `taskexit` statement in the following form:

$$\begin{aligned}
&\texttt{taskexit}(name_1 : \texttt{set } flagname_{11} \texttt{ to } bool_{11},\ldots,\\
&\quad \texttt{set } flagname_{1k_1} \texttt{ to } bool_{1k_1}, \texttt{add } tagname_{11},\ldots,\\
&\quad \texttt{add } tagname_{1l_1}, \texttt{clear } tagname'_{11},\ldots,\\
&\quad \texttt{clear } tagname'_{1m_1},\ldots,\\
&\quad name_n : \texttt{set } flagname_{n1} \texttt{ to } bool_{n1},\ldots,\\
&\quad \texttt{set } flagname_{nk_n} \texttt{ to } bool_{nk_n}, \texttt{add } tagname_{n1},\ldots,\\
&\quad \texttt{add } tagname_{nl_n}, \texttt{clear } tagname'_{n1},\ldots,\texttt{clear } tagname'_{nm_n}).
\end{aligned}$$

To execute a `taskexit` statement, the implementation updates the task's flags and tags and then commits any changes the task has made to the heap, the flags, or tag bindings. We define the transition for the `taskexit` statement below:

$$\begin{aligned}
&\langle\langle m,\mathcal{F},\mathcal{T},\tau,\mathcal{I}_{failed}\rangle,I,\langle m',\mathcal{F}',\mathcal{T}',\tau',L',\mathcal{S}'\rangle\rangle\\
&\quad \Rightarrow \langle\langle m',\mathcal{F}'',\mathcal{T}'',\tau',\mathcal{I}_{failed}\rangle,\emptyset,\emptyset\rangle,
\end{aligned}$$

$$\begin{aligned}
\texttt{where}\quad \mathcal{F}'' &= setflag(o_n,flagname_{nk_n},bool_{nk_n},\ldots\\
&\quad setflag(o_n,flagname_{n1},bool_{n1},\ldots\\
&\quad setflag(o_1,flagname_{1k_1},bool_{1k_1},\ldots\\
&\quad setflag(o_1,flagname_{11},bool_{11},\mathcal{F}')\ldots)\ldots)\ldots),\\
\mathcal{T}'' &= settag(o_n,L'(tagname_{nl_n}),\ldots\\
&\quad settag(o_n,L'(tagname_{n1}),\ldots\\
&\quad settag(o_1,L'(tagname_{1l_1}),\ldots\\
&\quad settag(o_1,L'(tagname_{11}),\mathcal{T}''_{clear})\ldots)\ldots)\ldots),\\
\mathcal{T}''_{clear} &= cleartag(o_n,L'(tagname'_{nm_n}),\ldots\\
&\quad cleartag(o_n,L'(tagname'_{n1}),\ldots\\
&\quad cleartag(o_1,L'(tagname'_{1m_1}),\ldots\\
&\quad cleartag(o_1,L'(tagname'_{11}),\mathcal{T}')\ldots)\ldots)\ldots).
\end{aligned}$$

3.2.5 Tag Allocation

We next define the tag instance allocation transition. A new tag instance is created using a tag allocation statement of the form:

$$\texttt{tag } tagname = \texttt{new tag}(tagtype).$$

We define the transition for the `new tag` statement below:

$$\begin{aligned}
&\langle\mathcal{G},I,\langle m',\mathcal{F}',\mathcal{T}',\tau',L',\mathcal{S}'\rangle\rangle \Rightarrow \langle\mathcal{G},I,\langle m',\mathcal{F}',\mathcal{T}',\tau'',L'',\mathcal{S}'\rangle\rangle,\\
&\qquad \texttt{where } \tau'' = \tau' \cup \{\langle tag_{fresh},tagtype\rangle\},\\
&\qquad\qquad L'' = L' \cup \{\langle tagname,tag_{fresh}\rangle\}.
\end{aligned}$$

Note that tag_{fresh} represents a fresh tag instance.

3.2.6 Object Allocation

We next define the new object allocation transition. Without loss of generality, we express the `new` statement in the following form:

$$\texttt{new}\ name(params)(\texttt{set}\ flagname_1\ \texttt{to}\ true, \ldots, \texttt{set}\ flagname_n\ \texttt{to}\ true)(\texttt{add}\ tagname_1, \ldots, \texttt{add}\ tagname_m).$$

We define the transition for the new statement below:

$$\langle \mathcal{G}, I, \langle m', \mathcal{F}', \mathcal{T}', \tau', L', \mathcal{S}' \rangle\rangle \Rightarrow \langle \mathcal{G}, I, \langle m'', \mathcal{F}'', \mathcal{T}'', \tau', L', \mathcal{S}'' \rangle\rangle,$$
$$\texttt{where}\ \mathcal{F}'' = setflag(o_{fresh}, flagname_n, true, \ldots setflag(o_{fresh}, flagname_1, true, \mathcal{F}') \ldots),$$
$$\mathcal{T}'' = settag(o_{fresh}, L'(tagname_n), \ldots settag(o_{fresh}, L'(tagname_1), \mathcal{T}') \ldots),$$

where o_{fresh} is a freshly allocated object.

3.2.7 Task Failure

We next define the behavior when the implementation detects an error. When an error is detected, the implementation aborts the execution of the current task and then records the task and parameter objects that caused the failure in the failed task record. We define the transition for the task abort below:

$$\frac{\texttt{error detected}}{\langle\langle m, \mathcal{F}, \mathcal{T}, \tau, \mathcal{I}_{failed}\rangle, I, \mathcal{M}\rangle \Rightarrow \langle\langle m, \mathcal{F}, \mathcal{T}, \tau, \mathcal{I}_{failed} \cup I\rangle, \emptyset, \emptyset\rangle}.$$

4 Runtime System

The Bristlecone runtime is responsible for dispatching tasks, detecting errors, and recovering from errors.

4.1 Task Execution

Recall that the task specification contains guard expressions for all of the task's parameters and the runtime executes a task when parameter objects are available that satisfy these guards. We next discuss how our implementation efficiently performs task dispatch. A naive approach to task dispatch could potentially be very inefficient—a parameter's guard expression is quantified over all objects in the heap.

4.1.1 Parameter Sets

The runtime maintains a *parameter set* for each parameter of each task. A parameter set contains all of the objects that satisfy the corresponding parameter's guard. For each object type, the runtime precomputes a list of parameter sets that objects of this type can potentially be a member of. When a task exit changes an object's flag settings or tag bindings, the runtime updates that object's membership in the parameter sets by traversing the precomputed list of possible parameter sets for the class and evaluating whether the object satisfies the guard expression to be a member of the parameter set.

Bristlecone also uses the parameter sets as root sets for garbage collection. Objects in Bristlecone are 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 as determined by membership in a parameter set. Note that it is possible to write incorrect programs that leave objects in task queues (e.g., consider a two parameter task with tagged parameters, the program might only change one parameter object's flags, leaving the other parameter object in the queue). Section 5 presents a static analysis that helps developers identify these types of memory leaks.

4.1.2 Task Queue

A *task invocation* is a tuple that includes both a task and bindings for that task's object parameters and tag parameters. An *active task invocation* is a task invocation that satisfies all of the task specification's guards and can therefore safely be invoked by the runtime. The runtime maintains the *task queue* of all active task invocations and executes task invocations from this task queue.

Our implementation maintains a conservative approximation of the task queue—our implementation's task queue may contain a number of nonactive task invocations in addition to all of the active task invocations. When an object is added to a parameter set, the implementation generates all active task invocations that bind that object to the corresponding parameter and then adds these active task invocations to the task queue. When an object is removed from a parameter set, our implementation does not remove task invocations from the task queue. Instead, before the implementation executes a task invocation in the queue, the implementation verifies that the task invocation is still active. Verification is straightforward—the runtime simply checks that the parameters are still in the corresponding parameter sets and the tag is still bound to the specified parameter objects.

4.1.3 Iterators

We next describe how our implementation efficiently computes the set of active task invocations. Our implementation incrementally computes the set of new task invocations when a task changes an object's tag bindings or its flags. The key insight in our implementation is that tag bindings restrict how parameter objects may be grouped together into a task invocation and therefore the implementation can leverage tags to efficient prune many possible task invocations that do not satisfy tag guards. Consider that the `sendPage` task in a Web server may require both a `WebRequest` object and a `Socket` object tagged with the same `connection` instance as parameters. Our implementation uses the `connection` tag instances to prune the search space of possible task invocations to avoid exploring pairs of `WebRequest` and `Socket` objects that are not bound to the same `connection` tag.

The search is structured as a sequence of iterators over parameter objects or parameter tag instances. The implementation uses two iterator types: *object instance iterators* and *tag instance iterators*. Object instance iterators iterate over possible parameter objects in the corresponding parameter set that are compatible with all tag variable bindings made by previous tag iterators in the sequence. In general, we expect that relatively few objects will be bound to a given tag instance and relatively few tag instances will be bound to a given object. Our implementation exploits this expectation to optimize the object iterators: If the parameter has a tag guard with a tag variable that was bound by a previous tag iterator, the implementation optimizes the object iterator to only iterate over the objects bound to that tag instance. Tag iterators iterate over tag instances that are bound to objects chosen by previous iterators in the sequence.

Note that the order of the iterators can affect the size of the search space that the implementation explores to generate all active task invocations. Our implementation precomputes iterator orderings for each parameter of each task. The implementation uses the following ordering priority:

1. Tag iterators for tags bound to parameter objects that have already been iterated over have the highest priority. We expect that the set of iterated tag instances will be small and therefore tag bindings will substantially prune subsequent object iterations for parameters bound to the same tag variable.
2. Object iterators for parameters with tags that are bound by previous tag iterators.
3. Object iterators for parameters with tags that have not yet been iterated over.
4. Remaining object iterators have the lowest priority.

4.1.4 Task Execution Semantics

Tasks may fail either as a result of software errors, hardware failures, or user errors. If a task fails, it may leave data structures in inconsistent states. Further computation using these inconsistent data structures will likely have unpredictable and potentially catastrophic results. To avoid this problem, tasks in Bristlecone have transactional semantics—if a task fails, the Bristlecone runtime aborts the task's transaction.

Recall that a potential issue with the use of transactions in traditional programming languages is that after the system recovers to the previous point, the system may simply reexecute the same deterministic fault and the fault will cause the system to fail repeatedly in the same way. Bristlecone addresses this issue by using the flexibility provided by the task-based language to avoid reexecuting the same failure. The Bristlecone runtime records the combination of task and parameter assignments that caused the failure and uses this record to avoid reexecuting the failed combination task and parameter assignments. Instead, the runtime executes other tasks to avoid retriggering the same underlying fault.

4.2 Error Detection

Errors can cause the computation to produce incorrect results and corrupt data structures, potentially eventually causing the software system to perform unacceptably. Bristlecone uses runtime checks to detect errors, enabling the software system to adapt its execution. The Bristlecone runtime uses error detection routines to trigger recovery actions.

Bristlecone detects many types of software errors. For example, the Bristlecone compiler generates array bounds checks. These checks verify that the software system does not read or write past the end of arrays. The Bristlecone compiler also generates the necessary type checks for array operations and cast operations. These checks ensure that the dynamic types of objects do not violate type safety.

The runtime uses hardware page protection to perform null pointer checks. This is implemented by catching the segmentation fault signal from the operating system. These checks ensure that the software system does not attempt to dereference a null pointer or write values to the fields of a null pointer. The runtime also uses hardware exceptions to detect arithmetic errors, including division by zero. Native library routines also signal errors to the runtime. For example, if a software system attempts to send data over a closed network connection, the runtime will signal an error. Software errors can also cause a program to loop. Looping can prevent the software system from providing services. It is straightforward to support developer-provided task timeouts that the runtime can use to detect looping tasks.

Bristlecone includes a runtime assertion mechanism to ensure that the execution is consistent with respect to specified properties. The developer can simply write imperative code to check properties or can use the assertion mechanism to call external consistency checking code. This mechanism is intended to be used to ensure data structure consistency or to use techniques such as design by contract to detect higher level errors. The mechanism can be used in conjunction with JML assertions [29], data structure consistency specifications languages [14], [16], or other runtime checkable specifications.

4.3 Error Recovery

Bristlecone was designed to support reasoning about failures at the level of tasks. In Bristlecone, a task either successfully completes execution or does not execute at all. Bristlecone adds write barriers to all object and array write operations to ensure that backup copies of all objects modified by the transaction exist. If the transaction is aborted, the objects are restored from the backup copies.

A second issue with the current implementation is transactional I/O. One solution is to use a transactional I/O API that delays the effects of I/O operations until a task commits.

If Bristlecone detects an error, it simply fails the entire task and aborts the transaction to roll back the state affected by the failed task. This recovery strategy greatly simplifies reasoning about the state of the software system after a failure. Aborting the transaction ensures that a failure does not leave partially updated data structures in inconsistent states.

Many software errors are deterministic. If Bristlecone reexecutes a failed task on the same parameters in the same state, it is likely that the task will fail again due to the same error. Bristlecone addresses this issue by maintaining a record of failures. For each failure, this record contains the combination of the failed task and the parameter assignments that failed. Bristlecone uses this record to avoid reexecuting the same failures by checking reference equality of the task's parameters. The Bristlecone runtime then uses the object flags to determine which tasks can be executed even though part of the computation has failed. To better handle nondeterministic failures, the approach can be extended to automatically retry a failed task execution a few times. We note that after a failure, a failed object can remain in the task queue and never be garbage collected. We expect that, in practice, software systems will be mostly correct, and therefore, failures will be rare occurrences and only small amounts of memory will be leaked due to failures.

4.4 Debugging and Error Logging

While it is desirable for deployed Bristlecone software systems to make every effort to avoid failures, during the development phase this behavior can mask failures and therefore complicate the debugging process. To facilitate debugging, Bristlecone can be configured to fail-fast. The fail-fast mode ensures that developers will notice software errors during the development process. Moreover, it would be straightforward to have the runtime record the state of the objects that caused the task failure. This information could help with debugging many software errors.

Furthermore, both developers and system administrators often want to be aware of failures in deployed systems so that the underlying faults can be fixed. Bristlecone contains a logging mechanism that records both the task that failed and the type of error. This log ensures that developers and system administrators are aware of failures in Bristlecone software systems and give the developers a starting point for diagnosing the cause of the failure. In some cases, developers may wish to create a custom framework to communicate failure data. It would be possible to provide an API for querying the runtime system about failures.

5 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 one-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. We abstract the set of reachable flag states F using a set of flag state nodes N.

The set T of tasks represents the possible set of tasks. The set $P \subseteq T \times \mathbb{N}$ represents the possible set of combinations of tasks and parameter indices for the invocation of a task on an object. The set of edges $E \subseteq N \times P \times N$ represents the possible object flag state transitions. If task T can be invoked with its ith parameter object in the flag state f_1 and exits with this object in the flag state f_2, then there is an edge for a task T for its ith parameter from the flag state node n_1 for f_1 to the flag state node n_2 for f_2, The lack of an edge implies that a transition is prohibited.

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

5.1 New Object Analysis

We next describe how the analysis determines the initial states of all objects that a task can potentially allocate. The complication is that Bristlecone methods with tag parameters were designed to be polymorphic in the tag type 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 classes 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  $callers := \{\}$
2  $alloc := \{\}$
3  $Q := T \times \{\}$
4  while $Q$ is not empty
5     remove method or task $\langle m, b\rangle$ from $Q$
6     $tvmap := \{\}$
7     if $m$ is a task
8        for each tag $t$ with type $\tau$ declared in $t$'s declaration
9           add $\langle t, \tau\rangle$ to $tvmap$
10    else if $m$ is a method and set of tag bindings $b$
11       add bindings $b$ to $tvmap$
12    for each tag allocation t=new tag $\tau$ in $m$
13       add $\langle \texttt{t}, \tau\rangle$ to $tvmap$
14    for each method call $c$ in $m$
15       for each method $m'$ that $c$ could invoke
16          add $\langle m', b'\rangle$ to $Q$ where $b'$ contains the tag variables in
17               $m$ declaration bound to their corresponding types
18          add $\langle\langle m, b\rangle, \langle m', b'\rangle\rangle$ to $callers$
19    for each allocation site $a$ in $m$
20       Compute the initial flag state $f$ using $tvmap$ to look up
21          tag types
22       add $\langle m, f\rangle$ to $alloc$.
23 do
24    for each $\langle\langle m, b\rangle, \langle m', b'\rangle\rangle$ in $callers$
25       $alloc := alloc \cup (\{m\} \times alloc(m'))$
26 while $alloc$ changes
```

Fig. 9. New allocation analysis.

Fig. 9 presents pseudocode for the object allocation analysis. The analysis specializes each method with a calling context that contains a list of the types of all the tag parameters. The analysis explores the application's call graph starting with the set of tasks as its root set. This phase of the analysis computes the map $alloc$ from tasks and methods to the set of initial flag states for the objects allocated by the task or method.

The analysis is structured as a workset algorithm. The workset Q is initialized with the set of tasks in the application. The algorithm removes a tuple consisting of either 1) a task m and the empty set or 2) a method m and a map b from the method's tag parameters to the tag types for the calling context. The analysis then initializes the tag variable map $tvmap$ from tag variables to the corresponding tag types. For methods, the analysis initializes $tvmap$ from the method's tag context b and for tasks, the analysis initializes $tvmap$ from the declared tag parameters in the task declaration. The analysis then updates $tvmap$ with any tag allocation statements in the current method. Next, the analysis processes each call site in the method. When the analysis discovers a method call to a previously unseen method calling context, it adds that method calling context to the workset Q. Finally, the analysis processes all of the allocation sites in the method. The analysis uses the map $tvmap$ from tag variables to tag types to determine which tag types are bound to newly allocated objects. The combination of the tag type information and the initial flag settings is sufficient for the analysis to determine the flag states of all objects the method or task can allocated. It

```
1 $discovered := \{\}$
2 $Q = \{$ StartupObject with initialstate:=true $\}$
3 while $Q$ is not empty
4   remove flag state $f$ from $Q$
5   for each task $t$ in $T$
6     for each parameter $i$ of task $t$
7       if ($f$'s type is not the declared type $\tau_i$ of parameter $i$ or
8         a subtype of $\tau_i$ ||
9         $f$ does not satisfy flag guard expression of parameter $i$ ||
10        $f$ is not bound to all declared tags for parameter $i$)
11        goto 6
12      for each flag state $f'$ in $alloc(t)$
13        if $f'$ is not in $discovered$
14          add $f'$ to $discovered$ and $Q$
15      for each task exit $e$ in $t$
16        compute the state $f'$ of parameter $i$ after task exit
17        add $\langle f, \langle t, i \rangle, f' \rangle$ to $E$
18        if $f'$ is not in $discovered$
19          add $f'$ to $discovered$ and $Q$
```

Fig. 10. State transition analysis.

continues this process until it has processed all reachable tasks and method calling contexts.

Finally, the analysis computes the transitive closure to determine the initial states allocated by each task or method and all methods transitively called. The analysis topologically sorts the strongly connected components of the call graph. The computation processes the strongly connected components of 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. Note that all methods in a strongly connected component can potentially (transitively) allocate the same set of flag states. A simple analysis of the algorithm structure reveals that complexity of the analysis is linear in the number of tasks plus the number of method calling contexts.

5.2 State Transition Analysis

We next describe the second phase of the static analysis that computes the set of 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.

Fig. 10 presents pseudocode for the state transition analysis. The analysis is structured as a workset algorithm. The basic approach is to start with the initial state of the `StartupObject`, and then, to explore all of the flag states that can be reached through the task invocations.

The analysis initializes the `StartupObject` object in its initial state with its `initialstate` flag set to true. For each flag state, the analysis analyzes the task specifications in steps 5-10 to determine all of the tasks that an object with this flag state could potentially serve as a parameter. For each such task in steps 12-14, the state transition analysis uses the results of the new object analysis to determine the flag states of all 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. Finally, the analysis 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 modeling 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. Because the analysis must analyze each task in the context of each reachable flag state, the complexity of the analysis grows with the product of the number of tasks and the number of reachable flag states.

5.3 Automated Analysis of Flag State Transition Diagrams

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.

5.4 Task Diagrams

Depending on the task at hand, the developer may wish to view a coarser abstraction of the application. 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 application. 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 application 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.

6 Experience

We next discuss our experiences using Bristlecone to develop three robust software systems: a Web crawler, a Web server, and a multiroom chat server.

6.1 Methodology

We have implemented the Bristlecone compiler. Our implementation consists of approximately 22,400 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 precise stop-and-copy garbage collection. The source code for our compiler and runtime is available at http://demsky.eecs.uci.edu/bristlecone/. We ran the benchmarks on a MacBook with a 2 GHz Intel Core Duo processor, 1 GB of RAM, and Mac OS X version 10.4.8.

For each benchmark, we developed two versions: a Bristlecone version and a Java version. We designed the Java versions to tolerate faults by using threads to isolate components of the computation. Without using threads to provide fault tolerance, the Java versions would have halted with the first failure.

Our evaluation was designed to evaluate how robust each version of the benchmark applications was to the large class of faults that cause the faulty thread or task to perform an illegal operation. This fault class includes faults that cause null pointer dereferences, out of bound array index errors, failed assertions, failed data structure consistency checks, library usage errors, and arithmetic exceptions. Our evaluation simulated the effects of this fault class by randomly injecting halting failures.

We used the Bristlecone compiler to automatically insert failure injection code after each instruction. We used the Java front end of our compiler framework to compile and instrument the Java versions. The failure injection code takes three parameters at runtime: the number of instructions to execute before considering injecting a failure, the probability that a failure will be injected, and the total number of failures to inject. For each benchmark, we selected the number of failures, and then, set the failure probability to ensure that the normal execution of the benchmark would reach the set number of failures.

6.2 Web Crawler

The Web crawler takes an initial Uniform Resource Locator (URL) as input, visits the Web page referenced by the URL, extracts the hyperlinks from the page, and then repeats this process to visit all of the URLs transitively reachable from the initial URL.

The Bristlecone version contains four tasks. The `Startup` task creates a `Query` object to store the initial URL that was specified on the command line and creates a `QueryList` object to store the list of URLs that the Web crawler has extracted. The `requestQuery` task takes a newly created `Query` object as input, contacts the Web server specified by the `Query` object, and then requests the URL specified by the `Query` object. The `readResponse` task reads the data that are currently available on the connection, and then checks if the task has received the complete Web page. The `processPage` task extracts URLs from the Web page, checks the `QueryList` object to see if the crawler has seen this URL before, and then creates a `Query` object if the URL has not been seen before.

The Java version uses a pool of three threads to crawl Web pages. Each thread dequeues a URL from a global list of pages to visit, downloads the corresponding Web page, extracts URLs from the Web page, and then stores any URLs it has not seen before into the global list of pages to visit.

We evaluated the robustness of the Web crawler by developing both a workload and a failure injection strategy. Our workload consisted of a set of 100 Web pages that each contain three hyperlinks to other Web pages in the set. We used randomized failure injection to inject failures into the executions of the Web crawlers. We injected three failures into each execution with each instruction having a 1 in 426,000 chance of failing.

	Java	Bristlecone
Web Pages Crawled (out of 100)	6	91

Fig. 11. Summary of Web crawler benchmark results.

We performed 100 trials of the experiment on each version. For each trial, we measured how many Web pages the crawler downloaded. Fig. 11 presents the results of the Web crawler experiments. Without the injected failures, both versions download 100 Web pages. With the injected failures, on average, the Bristlecone version downloaded 91 out of 100 Web pages and the Java version downloaded 6 out of 100 Web pages. While most of the injected failures in the Bristlecone version only affect crawling a single Web page, failures that are injected into either the startup task or the processing of the initial Web page can affect crawling many Web pages. Such failures prevent the Bristlecone version from discovering the URLs of any further pages and significantly lowered the Bristlecone version's average number of crawled pages.

6.3 Web Server

The Web server benchmark contains features that are intended to model an e-commerce server. The Web server integrates features often found in application servers—it maintains an inventory of merchandize and supports requests to perform commercial transactions on this inventory, including adding new items, selling items, and printing the inventory.

The Bristlecone version contains six tasks. The `StartUp` task creates a `ServerSocket` object to accept incoming connections, creates a `Logger` object to log the connections, and creates an `Inventory` object to keep track of the current inventory of merchandize. The `AcceptConnection` task processes incoming connections and creates a `WebSocket` objects to manage each connection. The `ProcessRequest` task reads the data that are currently available from the incoming connection, and then checks if the task has received the complete request. When the complete request is available, the `ProcessRequest` task parses the request to determine whether the request is an e-commerce transaction or a simple file request.

The `Transaction` task processes e-commerce transaction requests. It first inspects the request to determine whether the request is to add new items to the inventory, to make a purchase, or to display inventory, and then performs the requested operation. For example, after receiving a purchase request, the task looks up the price of the item in the `Inventory` object, verifies that the item is available, and if so, decrements the inventory count for the item and adds the price of the item to the sales figure.

The `SendFile` task processes file requests. It opens the requested file, reads the file's contents, and writes the file's contents to the socket. The `LogRequest` task logs all of the requests to the log file.

The Java version of the Web server uses a thread to monitor for incoming connections. When a new connection

	Java	Bristlecone
Failures to serve Inventory Responses	4.5%	1.5%
Correct Inventory Responses	68.6%	100%
Failures to Serve Request	3.8%	2.2%
Failures to Log Request	3.9%	2.6%

Fig. 12. Summary of Web server benchmark results.

arrives, the server spawns a separate thread for that incoming connection. The server uses a global object to store the inventory values. We used this design to isolate failures in connection threads to that specific request as much as possible. Note that failures can potentially corrupt the shared state. Note that unlike the Bristlecone version of the Web server, a failure in a connection thread will prevent the server from performing any further operations for that connection including logging the request.

We evaluated the robustness of both versions of the Web server by developing both a workload and a failure injection strategy. Our workload simulated Web traffic to the server. Our workload consisted of a sequence of 4,400 transaction requests. Our failure injection strategy utilized the failure injection code described in the previous section.

We used failure injection to randomly inject 50 failures into the execution with a probability of injecting a failure after a given instruction of 1 in 2,100,000. We performed 200 trials on each versions. For each trial, we recorded whether the final inventory request was served, whether the final inventory was consistent, how many requests each version failed to serve, and how many requests each version failed to log.

Fig. 12 summarizes the results of the fault injection experiments with the Web server. The Java version failed to serve the inventory request in 4.5 percent of the trials, while the Bristlecone version failed to serve the inventory request in 1.5 percent, representing a threefold reduction in the number of failures to serve inventory requests. More importantly, while the Java version served correct inventory responses only 68.6 percent of the time, the Bristlecone version served the correct inventory response 100 percent of the time. The Java version failed to serve 3.8 percent of the Web requests and Bristlecone version failed to serve 2.2 percent of the Web requests, representing a 42 percent reduction in the failure rate. The Java version failed to log 3.9 percent of the Web requests and the Bristlecone version failed to log 2.6 percent of the Web requests, representing a 33 percent reduction in the failure rate.

6.4 Chat Server

The multiroom chat server benchmark accepts incoming connections, asks the user to create a new room or select an existing room, and then allows users to chat with other users in the same chat room. The Bristlecone version contains six tasks. The `StartUp` task creates a `ServerSocket` object to accept incoming connections and a `RoomObject` to manage the chat rooms. The `AcceptConnection` task processes incoming chat connections. It creates a `ChatSocket` object to manage this connection, and then sends a message to ask the user to select a chat room.

The `ReadRequest` task reads the user's chat room selection. It reads the currently available data from the incoming connection and checks if the chat server has received the complete chat room selection. When the complete room request has been received, the `ProcessRoom` task processes the request. If the requested room does not exist, it creates the requested chat room. It then adds the user to the requested chat room. The chat server stores the mapping of chat room names to the set of chat room participants and, for each room, maintains a list of participants in the corresponding room.

The `Message` task processes incoming chat messages and stores these message in a `Message` object. The `SendMessage` task then reads these `Message` objects, parses the messages, and then sends the messages to all of the participants in the chat room. Note that a problematic message or other error condition that causes the `SendMessage` task to fail will not prevent the server from processing future messages from the same connection.

The Java version of the chat server uses a thread to monitor for incoming connections. When a new connection arrives, the server spawns a separate connection thread for that incoming connection. The server uses a global object to store the set of chat rooms. Unless a failure corrupts the room list, this design isolates failures in connection threads to the specific connection. Note that unlike the Bristlecone version, a single failure in a connection thread will prevent the server from relaying any further messages from that connection.

We evaluated the robustness of both versions by developing a workload and a failure injection strategy. Our workload simulated multiple users chatting. Our workload sent a total of 800 messages. Our failure injection strategy utilized the failure injection code described in the previous section.

We used failure injection to randomly inject 10 failures into the execution with a probability of injecting a failure after a given instruction of 1 in 270,000. We performed 100 trials on each of the two versions. For each trial, we recorded how many messages were successfully transmitted.

In the presence of the injected failures, the Java version failed to deliver 39.9 percent of the messages and the Bristlecone version failed to deliver 19.3 percent of the messages, representing a factor of two reduction in the failure rate.

6.5 Experiences Writing Bristlecone Applications

We have developed Bristlecone and Java versions of three different benchmark applications. In general, we found writing Bristlecone applications to be straightforward. The process begins with identifying both the high-level operations and the key data structures in the program. Next, the developer determines the conceptual states of the key data structures and the dependencies between these conceptual states and the high-level operations. The developer then uses this information to write task specifications. We found that the Bristlecone versions often shared major components with our Java versions.

The Bristlecone versions of the benchmarks were approximately the same size as the Java versions. The Bristlecone version of the Web crawler contained 20 percent fewer lines of code than the Java version, the Bristlecone version of the Web server contained 2 percent more lines of code than the Java version, and the Bristlecone version of the chat server contained 5 percent more lines of code. The Bristlecone version of the Web crawler was shorter because it did not require an auxiliary data structure to store queries.

6.6 Performance

Although Bristlecone uses standard compilation techniques for the body of methods and tasks, it incurs extra overheads supporting transactions and task invocation. We have measured the current implementation's task invocation overhead on a microbenchmark to be 0.63 microseconds per task invocation on a 2.2 GHz Core 2 Duo. The same microbenchmark on our previous checkpointing-based implementation takes 4.68 microseconds per task invocation.

To measure the overhead of Bristlecone on a real computation, we ported a Mandelbrot set computation that generates a $3{,}200 \times 3{,}200$ image to C and Bristlecone. The Bristlecone version divides the computation into 800 tasks. The Bristlecone version took 4.82 seconds and the C version took 3.95 seconds, representing a slowdown of 22 percent relative to C. Note that a number of opportunities remain to optimize the overhead of task invocation and eliminate many of the write barriers.

6.7 Discussion

Our experience indicates that software systems developed using Bristlecone can recover from many otherwise fatal failures. The Bristlecone versions of all three benchmarks were able to recover from many more injected failures and provided a higher quality of service than the hand-designed Java versions.

Note that these results only hold for software faults that can be automatically detected. These results can be generalized to include faults that cause the application to silently perform an incorrect action if the developer provides Bristlecone with a runtime-checkable correctness specification that detects the error. Examples of such specifications include runtime assertions or data structure consistency specifications.

7 Analysis Experience

We next discuss our experiences using the flag state analysis tool to explore the behavior of several Bristlecone programs. We report our experience for: TTT, a tic-tac-toe game; a Web server; and a chat server. In these experiments, the coauthor using the static analysis tool had no prior experience with the benchmark programs but was experienced with Bristlecone.

7.1 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, TTT did not allow us to complete the game and it printed out multiple copies of the board after each move.

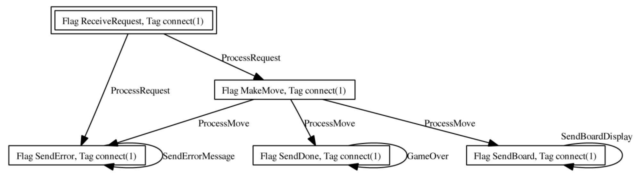
Flag ReceiveRequest, Tag connect(1)
ProcessRequest
ProcessRequest
Flag MakeMove, Tag connect(1)
ProcessMove
ProcessMove
ProcessMove
SendBoardDisplay
Flag SendError, Tag connect(1)
SendErrorMessage
Flag SendDone, Tag connect(1)
GameOver
Flag SendBoard, Tag connect(1)

Fig. 13. Flag state transition diagram with `ReceiveRequest` reset.

We used the flag state analysis to understand TTT's erroneous behavior. 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.

Fig. 13 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 tasks may be executed repeatedly, causing the server to display multiple copies of the same board, print error messages many times, and print the exit 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.

Fig. 14 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

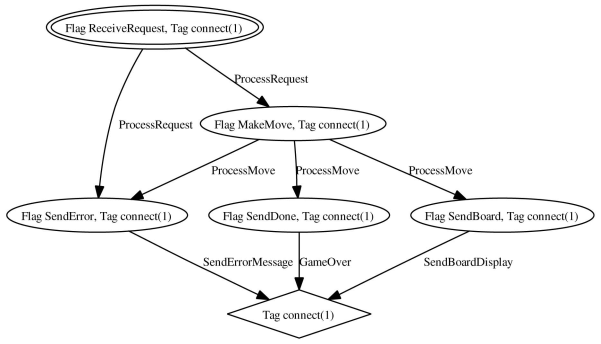
Flag ReceiveRequest, Tag connect(1)
ProcessRequest
ProcessRequest
Flag MakeMove, Tag connect(1)
ProcessMove
ProcessMove
ProcessMove
Flag SendError, Tag connect(1)
Flag SendDone, Tag connect(1)
Flag SendBoard, Tag connect(1)
SendErrorMessage
GameOver
SendBoardDisplay
Tag connect(1)

Fig. 14. Flag state transition diagram with multiple invocation correction.

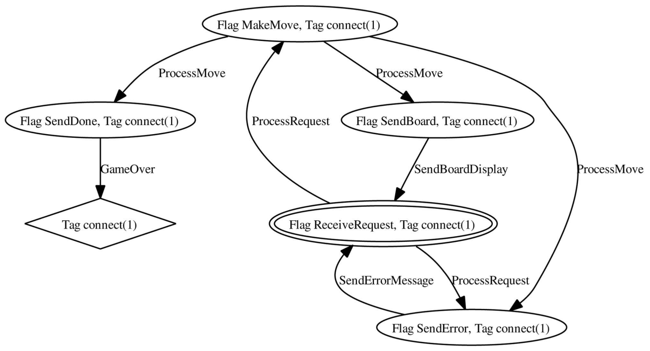
Flag MakeMove, Tag connect(1)
ProcessMove
ProcessMove
Flag SendDone, Tag connect(1)
ProcessRequest
Flag SendBoard, Tag connect(1)
GameOver
SendBoardDisplay
ProcessMove
Tag connect(1)
Flag ReceiveRequest, Tag connect(1)
SendErrorMessage
ProcessRequest
Flag SendError, Tag connect(1)

Fig. 15. Final flag state transition diagram.

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. Fig. 15 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.

7.2 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.

Fig. 16 presents the flag state transition diagram for the `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.

Fig. 17 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.

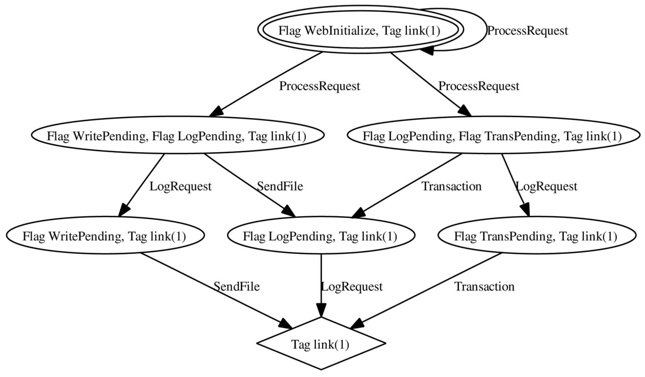
Flag WebInitialize, Tag link(1)
ProcessRequest
ProcessRequest
ProcessRequest
Flag WritePending, Flag LogPending, Tag link(1)
Flag LogPending, Flag TransPending, Tag link(1)
LogRequest
SendFile
Transaction
LogRequest
Flag WritePending, Tag link(1)
Flag LogPending, Tag link(1)
Flag TransPending, Tag link(1)
SendFile
LogRequest
Transaction
Tag link(1)

Fig. 16. Flag state transition diagram for the WebServerSocket class.

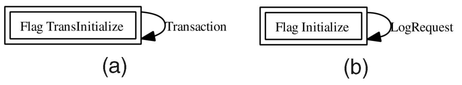

Fig. 17. Flag state transition diagrams for the inventory and logger classes.

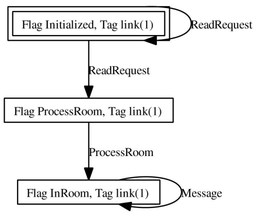

Fig. 18. Flag state transition diagram for the ChatSocket class.

7.3 Chat Server

The multiroom 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.

Fig. 18 presents the flag state transition diagram for the `ChatSocket` class. The presence of rectangular nodes indicates 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

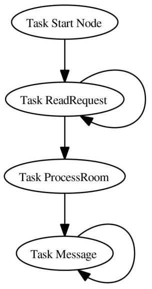

Fig. 19. Task diagram for the ChatSocket class.

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 coarser abstraction of an application's behavior. The task diagram is designed to further abstract the interaction patterns between tasks and objects. Fig. 19 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.

7.4 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 a 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 paper, 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 who did not have access to any Bristlecone tutorial, assistance from Bristlecone developers, or the static analysis developed in the paper.

8 Related Work

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

8.1 Approaches to Reliable Software

The standard approach to dealing with software failures is to work hard to find and eliminate software faults. Approaches such as extensive testing [9], static analysis [20], [39], software model checking [13], error correction codes [41], and software isolation mechanisms [1] are all designed, in part, to eliminate as many potential errors as possible. We expect that Bristlecone will complement these techniques by enabling software systems to recover from software errors that the other techniques miss.

Many programming languages, including Java, provide an exception handling mechanism [21]. Writing exception handlers requires developers to reason about what parts of the computation are effected by the failure and how to recover the computation from a failure. Bristlecone eliminates this difficulty by automatically reasoning about program dependencies and generating the appropriate recovery actions.

Backward recovery uses a combination of checkpointing and acceptance tests (or error detection) to prevent a software system from entering an incorrect state [48], [37], [11], [47]. Unfortunately, it can be difficult to handle deterministic failures using backward recovery as the same software error will likely cause the software system to repeatedly fail. Forward recovery uses multiple copies of a computation to recover from transient errors [26]. Forward recovery is designed to handle intermittent failures—it cannot help deterministic errors that affect all copies of the computation. Bristlecone's task specifications allow Bristlecone applications to continue to perform useful work even in the presence of deterministic failures by executing the parts of the application that do not depend on the failure.

Fault tolerance researchers have developed many methods to address software failures. Recovery blocks allow a developer to provide multiple implementations of an algorithm and an acceptance test for these implementations [4]. This technique requires the developer to expend the effort to develop multiple implementations and acceptance tests. Furthermore, the recovery block technique may fail if the algorithms share a common defect or if there is an error in the acceptance test. In N-version programming, the developer constructs a software system out of multiple, independent implementations and a decision algorithm to decide which result to use in the event of a disagreement [6]. For many applications, the cost of developing multiple independent implementations is prohibitive. Bristlecone enables these applications to benefit from improved reliability at a lower cost.

The Recovery-Oriented Computing project has explored integrating an undo operation into software systems [36] and constructing systems out of a set of individually rebootable components [10]. One issue with recursive restartability is that key data structures can be corrupted causing the system to fail. Bristlecone has been designed to partition data to minimize how much data a failure can affect.

Researchers have used metalanguages to decompose numerical computations into parallelizable tasks [38]. This technique is applicable to parallelizable numerical computations that compute many subproblems and then combine the subproblem results to compute an overall result. If one of the subcomputations executes slowly, this approach can ignore the subcomputation. Bristlecone is designed to handle a broader class of systems including servers, control systems, and office applications and can provide stronger correctness guarantees.

8.2 Related Languages

A previous version of this paper appeared in ECOOP 2008 [15]. This version of the publication adds formal semantics for the language, a static analysis of the Bristlecone task specifications, and two graphical representations of this analysis's results. These diagrams graphically characterize the set of transitions that tasks can perform on the objects in the application. The current implementation has been modified to use write barriers to only copy modified objects instead of checkpointing the entire heap that is reachable from the current task's parameters.

A key component of Bristlecone is decoupling unrelated conceptual operations and tracking data dependencies between these operations. Bristlecone's approach contains common elements with many parallel programming paradigms [8]. Data flow computation was one of the earlier computational models that keeps track of data dependencies between operations so that the operations can be parallelized [27]. Note that data flow languages are not designed to handle failures—a failure in a data flow program will likely cause an operation to fail to place a value in a queue, which would likely cause the application to fail catastrophically because operations that operate on multiple queues would pair the wrong values for the rest of the computation. Bristlecone ensures that failures cannot cause the wrong parameter objects to be paired together or prevent a task from operating on parameter objects that were not affected by the error.

Tuple space languages, such as Linda [19], decouple computations to enable parallelization. The threads of execution communicate through a set of primitives that manipulate a global tuple space. While these systems were not designed to address software errors as errors in these systems can permanently halt the execution of threads, Bristlecone implements a similar technique to decouple the execution of its tasks.

The orchestration language Orc [12] 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 track 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], [2]. Actors were 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 a task does not affect future invocations of that task on different inputs.

The Argus programming language organizes processes under guardians and isolates process failures to the guardian under which it executes [30]. Inconsistencies could potentially cause the enclosing guardian to shut down. Argus supports failure recovery 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 computations as a set of tasks [42], [31]. Tasks are created and destroyed by the program. A task becomes reducible (executable) once the constraint store satisfied the task's guard. Task reducibility is monotonic—once a task is reducible, it is always reducible. Task activation in Bristlecone is not monotonic—the developer can temporarily disable a task when other tasks have placed objects into states that are incompatible with the task or when the effect of a task is no longer desirable. Nonmonotonicity makes it straightforward for a Bristlecone application to use 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 tasks.

Concurrent Prolog is logic-based language that uses unification to prove a goal [40]. The proof corresponds to the program's execution. 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 backward. Concurrent Prolog programs may be able to recover from some failures by finding a different execution that reaches the same goal. The downside is that if a failure prevents the program from completely achieving its goal, the program will be unable to make partial progress. Bristlecone works forward, and therefore, 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 set of supervisors and a hierarchy of increasingly simple implementations of the same functionality [5]. The supervisors monitor the computation for errors. If an error is detected, the system falls back to a simpler implementation in the hierarchy. Ericsson has taken this approach in their telephone switches. Bristlecone is complementary to the supervisor approach—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 development effort and could potentially make recovery cost effective for a larger set of applications. 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.

The properties that the flag state analysis extracts are related to typestate properties [43]. Nanda et al. [34] present a static analysis that infers typestate properties. Ammons et al. [3] present a dynamic analysis that monitors function calls to a component to automatically extract finite-state machines that describe the component interface. Wasylkowksi et al. [44] present a static analysis that can extract and check finite-state automata for method calls on individual objects. Whaley et al. [46] combine static analysis and dynamic analysis to discover finite-state machine component interface models.

Bristlecone separates information about the structure of high-level components of a software systems and how these components interact from the low-level code that implements the components. Bristlecone separation of these different program aspects shares philosophical underpinnings with aspect-oriented programming [28]. Viewed in this light, Bristlecone's contribution can be seen as a mechanism to separate high-level structural aspects from low-level imperative code.

8.3 Related Software Architectures

The staged event-driven architecture (SEDA) pushes events through stages [45]. Note that this architecture was designed for high-performance computation and not fault tolerance. An error in a stage can prevent relaying the event and cause information to be lost. Stages also have local state; therefore, corruption of this state will cause that stage to shutdown until reboot. It appears difficult to specify that an application should either execute one sequence of operations or a second sequence, but not both.

9 Conclusion

We have successfully developed several robust software systems using Bristlecone. Bristlecone software systems consist of a set of interacting tasks, with each task implementing one of the conceptual operations in the software system. The developer specifies how these tasks interact using task specifications. Bristlecone uses transaction to recover data structures from task failures. Bristlecone then uses task specifications to reason about how to continue execution in the presence of a failed task. The key results in this paper include the Bristlecone language, the Bristlecone compiler and runtime, a static analysis of Bristlecone task specifications, and our experience using the Bristlecone language. Our experience indicates that the task-based approach used in Bristlecone can effectively enable software systems to recover from otherwise fatal errors. Bristlecone promises to increase the robustness of software systems and to decrease the cost of developing many classes of robust software systems.

Acknowledgments

This is a revised and extended version of the paper that appeared in ECOOP 2008 [15]. This research was supported by the US National Science Foundation under Grant Nos. CCF-0725350 and CCF-0846195.

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," *Proc. USENIX Summer Conf.*, 1986.

[2] G. Agha, I.A. Mason, S.F. Smith, and C.L. Talcott, "A Foundation for Actor Computation," *J. Functional Programming,* vol. 7, no. 1, pp. 1-72, 1997.

[3] G. Ammons, R. Bodik, and J.R. Larus, "Mining Specifications," *Proc. 29th Ann. ACM Symp. Principles of Programming Languages,* 2002.

[4] T. Anderson and R. Kerr, "Recovery Blocks in Action: A System Supporting High Reliability," *Proc. Second Int'l Conf. Software Eng.,* pp. 447-457, 1976.

[5] J. Armstrong, "Making Reliable Distributed Systems in the Presence of Software Errors," PhD thesis, Swedish Inst. of Computer Science, Nov. 2003.

[6] A. Avizienis, "The Methodology of n-Version Programming," 1995.

[7] W.O. Baker, I.M. Ross, J.S. Mayo, and D.C. Stanzione, "Bell Labs Innovations in Recent Decades," *Bell Labs Technical J.,* vol. 5, no. 1, pp. 3-16, Jan.-Mar. 2000.

[8] N. Benton, L. Cardelli, and C. Fournet, "Modern Concurrency Abstractions for C#," *Proc. 16th European Conf. Object-Oriented Programming,* 2002.

[9] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing Based on Java Predicates," 2002.

[10] G. Candea and A. Fox, "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel," *Proc. Workshop Hot Topics in Operating Systems,* pp. 110-115, May 2001.

[11] K.M. Chandy and C. Ramamoorthy, "Rollback and Recovery Strategies," *IEEE Trans. Computers,* vol. 21, no. 2, pp. 137-146, Feb. 1972.

[12] W.R. Cook, S. Patwardhan, and J. Misra, "Workflow Patterns in Orc," *Proc. 2006 Int'l Conf. Coordination Models and Languages,* 2006.

[13] 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," *Proc. 2000 Int'l Conf. Software Eng.,* 2000.

[14] B. Demsky, C. Cadar, D. Roy, and M.C. Rinard, "Efficient Specification-Assisted Error Localization," *Proc. Second Int'l Workshop Dynamic Analysis,* 2004.

[15] B. Demsky and A. Dash, "Bristlecone: A Language for Robust Software Systems," *Proc. 2008 European Conf. Object-Oriented Programming,* 2008.

[16] B. Demsky and M. Rinard, "Data Structure Repair Using Goal-Directed Reasoning," *Proc. 2005 Int'l Conf. Software Eng.,* May 2005.

[17] M.D. Ernst, A. Czeisler, W.G. Griswold, and D. Notkin, "Quickly Detecting Relevant Program Invariants," *Proc. 22nd Int'l Conf. Software Eng.,* June 2000.

[18] D. Garlan and D. Notkin, "Formalizing Design Spaces: Implicit Invocation Mechanisms," *Proc. Fourth Int'l Symp. VDM Europe on Formal Software Development—Volume I,* pp. 31-44, 1991.

[19] D. Gelernter, "Generative Communication in Linda," *ACM Trans. Programming Languages and Systems,* vol. 7, no. 1, pp. 80-112, 1985.

[20] 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," *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* 1996.

[21] J.B. Goodenough, "Structured Exception Handling," *Proc. Second ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages,* 1975.

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

[23] T. Harris, "Exceptions and Side-Effects in Atomic Blocks," *Science of Computer Programming,* vol. 58, no. 3, pp. 325-343, 2005.

[24] G. Haugk, F. Lax, R. Royer, and J. Williams, "The 5ESS(TM) Switching System: Maintenance Capabilities," *AT&T Technical J.,* vol. 64, no. 6, pp. 1385-1416, July/Aug. 1985.

[25] C. Hewitt and H.G. Baker, "Actors and Continuous Functionals," technical report, Massachusetts Inst. of Technology, 1978.

[26] K. Huang, J. Wu, and E.B. Fernandez, "A Generalized Forward Recovery Checkpointing Scheme," *Proc. 1998 Ann. IEEE Workshop Fault-Tolerant Parallel and Distributed Systems,* Apr. 1998.

[27] W.M. Johnston, J.R.P. Hanna, and R.J. Millar, "Advances in Dataflow Programming Languages," *ACM Computing Surveys,* vol. 36, no. 1, pp. 1-34, 2004.

[28] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. 11th European Conf. Object-Oriented Programming,* pp. 220-242, 1997.

[29] G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs, "JML: Notations and Tools Supporting Detailed Design in Java," *Proc. Conf. Object Oriented Programming Systems Languages and Applications,* pp. 105-106, 2000.

[30] 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 Inst. of Technology, Nov. 1987.

[31] M. Mehl, "The Oz Virtual Machine—Records, Transients, and Deep Guards," PhD thesis, Technische Fakultät der Univ. des Saarlandes, 1999.

[32] B. Meyer, "Design by Contact," *Computer,* vol. 23, no. 10, pp. 40-51, Oct. 1992.

[33] S. Mourad and D. Andrews, "On the Reliability of the IBM MVS/XA Operating System," *IEEE Trans. Software Eng.,* vol. 13, no. 10, pp. 1135-1139, Oct. 1987.

[34] M.G. Nanda, C. Grothoff, and S. Chandra, "Deriving Object Typestates in the Presence of Inter-Object References," *Proc. 20th Ann. ACM SIGPLAN Conf. Object Printed Programming, Systems, Languages, and Applications,* 2005.

[35] V.S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An Efficient and Portable Web Server," *Proc. USENIX 1999 Ann. Technical Conf.,* pp. 199-212, 1999.

[36] 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, Univ. of California, Berkeley, Computer Science, Mar. 2002.

[37] J.S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," *Proc. Usenix Winter Technical Conf.,* pp. 213-223, Jan. 1995.

[38] M. Rinard, "Probabilistic Accuracy Bounds for Fault-Tolerant Computations That Discard Tasks," *Proc. 20th ACM Int'l Conf. Supercomputing,* 2006.

[39] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric Shape Analysis via 3-Valued Logic," *Proc. Symp. Principles of Programming Languages,* pp. 105-118, 1999.

[40] E. Shapiro, "The Family of Concurrent Logic Programming Languages," *ACM Computing Surveys,* vol. 21, no. 3, pp. 413-510, 1989.

[41] P.P. Shirvani, N.R. Saxena, and E.J. McCluskey, "Software-Implemented EDAC Protection against SEUs," *IEEE Trans. Reliability,* vol. 49, no. 3, pp. 273-284, Sept. 2000.

[42] G. Smolka, "The Oz Programming Model," *Proc. European Workshop Logics in Artificial Intelligence,* p. 251, 1996.

[43] R.E. Strom and S. Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability," *IEEE Trans. Software Eng.,* vol. 12, no. 1, pp. 157-171, Jan. 1986.

[44] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting Object Usage Anomalies," *Proc. Sixth Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 35-44, 2007.

[45] M. Welsh, D.E. Culler, and E.A. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," *Proc. 18th Symp. Operating Systems Principles,* Oct. 2001.

[46] J. Whaley, M.C. Martin, and M.S. Lam, "Automatic Extraction of Object-Oriented Component Interfaces," *Proc. 2002 ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* 2002.

[47] J.W. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Comm. ACM,* vol. 17, no. 9, pp. 530-531, 1974.

[48] Y. Zhang, D. Wong, and W. Zheng, "User-Level Checkpoint and Recovery for LAM/MPI," *ACM SIGOPS Operating Systems Rev.,* vol. 39, no. 3, pp. 72-81, 2005.

Brian Demsky received the PhD degree in 2006 from the Massachusetts Institute of Technology (MIT) while with the Program Analysis and Compilation Group at the MIT Computer Science and Artificial Intelligence Laboratory. He is an assistant professor in the Department of Electrical Engineering and Computer Science at the University of California, Irvine. His current research interests include software reliability, software engineering, compilation, software debugging, and program understanding.

Sivaji R. Sundaramurthy received the MS degree in 2008 from the University of California, Irvine, while working with Dr. Demsky in the area of software engineering and program understanding. Currently, he is a software developer with the Global Financial Systems Group at Amazon.com. His research interests, albeit dormant, include parallel programming, algorithms, and program understanding.