Detecting Data Races on Framework-Based Applications

Federico Balaguer, Thuc S. M. Ho , Ralph Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {balaguer,mauhosi2,johnson}@uiuc.edu

Abstract— **Race conditions are hard to detect in framework-based applications. Frameworks often improve performance by providing threading, but this threading is usually hidden from application programmers. Therefore, it is easy for application programmers to accidentally create data races.**

Data races can be detected tracing the flow of execution, but tracing tends to produce too much data. However, the structure of frameworks can be used to control the amount of data collected and that makes tracing practical.

We have developed a tracing and analysis tool that allows application programmers to explore different configurations of an application and find probable data races. This enables them to have mutithreaded frameworks safely without having to learn the details of the framework design.

I. Introduction

Multi-threading programming has increased along with the popularity of web-services and server-applications. Multi-threading programming is used on the server side for handling connections and in the client side for performing background operations [15]. Threads provide great benefits but they are hard to use and debug as they make the expected behavior of code less apparent [4].

Threads can affect the behavior of applications developed with frameworks. Frameworks are an object-oriented reuse technique and are widely used in many application domains [11], [16]. Frameworks provide template execution paths (flow of control) that eventually call code provided by the application [8]. Although the framework assumes it is in control, it relies at some point on the code provided by the application. This code can use other frameworks which in turn will assume control over the execution. Between each framework lies the code of the application that has to fill the gaps among frameworks. Figure 1 shows the code of the operation doPost() that will be called from a servlet framework. Inside this operation another framework is used to retrieve from and save objects to a database.

```
doPost(HttpServletRequest req, HttpServletResponse rsp)
// Pseudo code
//retrieve or create account
account =
WebAccount.factory(req.getParameterValues(”ACC”));
//update account
account.update(req);
//save in db (insert / update)
writeResponse(account.save(), rsp);
```

Fig. 1. doPost() operation of a servlet framework

There are many factors that can determine whether this piece of code has a data race. First, different Http requests can be handled with just one servlet or a new servlet can handle each of them. Second, the persistence framework can make blocking or non-blocking calls to the database. Third, queries and SQL commands can be performed outside or inside transactions, moreover the transaction semantics can vary. A data race is a sign that the chain of execution between frameworks is broken. Data races are a non-atomic execution of critical sections [21]. We found cases where data races exist even when the observable results in the database were correct. By changing the database settings we were able to produce inconsistencies in the database.

This paper presents a technique for finding data races on applications developed using multiple frameworks. Our approach has two steps. First, traces are collected from the application execution. The traces are represented as execution trees [17] and are collected at those points where the frameworks hands control over to the application code. Second, traces are used to compute a set of conflicting accesses using the LockSet algorithm as described in [26]. von Praun et al showed that conflicting accesses are good approximations to data races [28].

A. Frameworks Collision

Frameworks are usually designed under the assumption that they are in control of the execution of the application using them [16]. Berlin showed that inconsistencies can appear in an application when multiple frameworks implement their own event-loop [3]. Garlan et al described how a user-interface framework and a network messaging framework interfere with each other; when one of the frameworks is in control the other one loses events [12].

The problem we are describing is different. One framework that has control over the execution of an application hands control over to another framework that does not fulfill its expectations. For example, servlets such as the one in Figure 1 are implemented out of web frameworks that eventually call the `doPost()` or `doGet()` functions. The servlet frameworks are responsible for providing the multi-threading model in which one instance of a servlet can service one connection (monothreading) or multiple connections (multithreading).

Usually developers of servlets have no control over the computation before the servlets are called and the only opportunity they have to implement functionality is inside `doPost()` and `doGet()`. The servlet in Figure 1 retrieves an object from a database, updates fields in the account object and finally stores the object back into the database using some persistent framework. Usually, persistence frameworks take care of database connection, sessions and transactions and allow developers to customize the mapping between objects and tables. Problems can arise if the persistence framework makes non-blocking calls to the database because that could unexpectedly yield control of the servlet allowing other servlets to run.

One alternative to minimize the possibility of data races in framework-based applications is to create critical sections at the point where the framework calls the application code. For many frameworks this is impractical because of the restrictions it imposes on applications. Another alternative is to force developers to run the code that interacts with the framework in only one thread. Swing uses this solution and experience shows that it is difficult to program [24]. No matter the solution is based on critical sections, semaphores or advanced thread programming, developers need to find the source of the problem. Our approach allows developers to explore different configurations of the system and study the effect that each configuration has on shared state.

B. Manipulating Traces

Lange and Nakamura presented several techniques to display the execution information of programs [18]. These techniques are based on Execution Trees. Each tree represents the execution of a function that is the root of the tree [23]. Figure 2 shows an execution tree representing the execution of `doPost()` of the servlet in Figure 1.

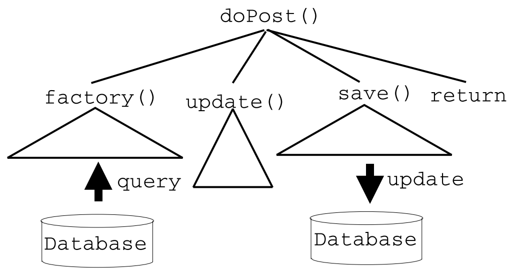


Fig. 2. Execution Tree of operation doPost()

Nodes in the tree can be one of five possible node types:

- **Function Invocation** nodes represent the invocation of primitive functions or messages sent to some object. When a function invocation is root of a tree, the branches represent the execution of that function.
- **Variable Access** nodes represent assignments and reads to a variable.
- **Control Primitive** nodes represent nodes that give up control such as yield, and semaphore operations.
- **Return** nodes represent the end of the execution of function invocation, in normal conditions the rightmost node of a tree is a return node
- **Jump** nodes represent loops and if-then-else structures

The sequence of execution for one tree can be generated by doing the pre-order traverse of the nodes, that is, visiting first the root and then recursively visiting each of the child nodes.

Execution trees scale to multi-threaded programs but a broader context is needed since one execution tree is only responsible for what is done by one thread. Multi-threaded programs are represented by a set of competing execution trees that gain and relinquish control over the execution. Once the execution trees are established for each thread a tool can automatically produce a sequence of execution among the threads and also the sequence in which objects and their variables are accessed.

II. Finding Conflicting Accesses

A. LockSet algorithm

Savage et al presented the LockSet algorithm as part of Eraser, a tool for finding data races [26]. LockSet searches for shared-memory references that don't have a consistent locking behavior.

Given a non empty set of execution trees that represent threads of a program, it is possible to find possible conflicting accesses by computing the set of variable accesses. Each variable access carries the following information:

- `tid`: identification of the thread
- `vid` the identification of the variable being accessed. The identification of the variable is a composed key with objectID and the index of the variable inside the containing object.
- `kind`: the kind of access, it can be either `read` or `write`.
- `fid` the identification of the function issuing the access
- `pc` the program counter
- `SemSet`: the set of semaphores acquired (but not released) by the thread until this point in the execution.

A conflicting access happens when in the set of variable access at least two tuples access the same variable, are generated by different threads, at least one of them is a write and intersection of acquired locks is empty.

B. Green LockSet

LockSet is a simple algorithm and it gives too many false positives. Eraser implements a modified version of LockSet that consider the following exceptions: initialization, read-share data and read-write locks. Another source of improvement can be the thread scheduling policy. There are widely accepted scheduling policies based on non-preemptible cooperative threads i.e. fibers that are used in many object-oriented systems. Fibers, also known as green threads, are thread libraries implemented at the process level. They are independent of the operating system and primarily implemented in user-space. In fibers-based scheduling a thread yields control voluntarily by calling a thread primitive such as `suspend` or implicitly by making non-blocking call. This is important because the lack of time-slicing makes the execution of the system possible to simulate. Figure 3 shows an execution tree of our WebAccount servlet example. The tuple $< t1, v1... >$ represents the access to the variable $v1$ by the thread $t1$. It is possible to infer that the servlet will yield control twice before returning control to the servlet framework, if it is using non-blocking calls to the database (I/O). However, if the servlet is using blocking calls, it will not yield control and then the access tuple $< t1, v1... >$ is unique for the variable v_1.

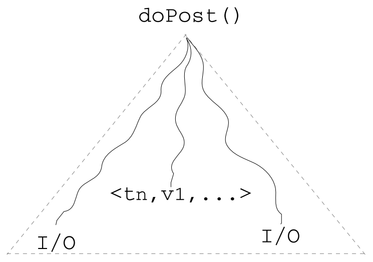


Fig. 3. Execution Tree of operation doPost() with I/O

We extend the LockSet algorithm with the semantics of green threads. Given a collection of ExecutionTrees the resulting Green LockSet algorithm visits the nodes in each tree following green thread scheduling policy. The algorithm reacts to the types of node as follows:

- For each node representing a `function call`, it visits the children of that node.
- For each node representing a `variable access`, it adds the access to the set of variable accesses. The algorithm checks for LockSet's conflicting accesses.
- For each node representing an operation that `control primitives`, it checks if the operation yields control or is a semaphore operation. In the first case, the algorithm picks the next-to-run thread. For operations acquiring or releasing a semaphore it computes the state of the semaphore and proceeds accordingly.
- For each node representing a `return`, it checks whether the thread ended its computation. When a thread finishes the algorithm removes all accesses added to the set of variable accesses
- For each node representing a `jump`, it visits the `jump`'s subtree accordingly.

The set of operations that are considered control primitives can be changed making simple blocking operation into a non-blocking one. This in turn will change the way variables are accessed in the execution tree.

III. Implementation

A. Tracing Tool

Our current design of the tracing tool contains the following main components: tracers, tracing service, trace logs, and a trace viewer. They are built upon the concept of activation points that keep track of tracing information.

An activation point is an abstraction representing the set of information collected by the tracer after each stepping activity. For stack-based virtual machines such as the Java Virtual Machine or the Smalltalk virtual machine in VisualWorks, activation points can populate its internal data structure from the stack frame (the method invocation context right after a step). In our implementation (Figure 4), an activation point stores the compiled code, byte code index, the class implementing the method, the receiver's actual class, the selector associated with the compiled code (if one exists) and the hash of the receiver object as an object identity. For child processes (threads) activation points have an extra reference to the tracer attached to the child process. This allows connecting the parent trace log with the child trace log.

Activation points also keep track of types of activities being performed (a message sending, a jump, a variable assignment, a variable read, a loop or a return from method invocation). They are kept in the `action` list in figure 4. This information is useful while analyzing the trace log contents. Part of these information can be generated by running additional tools after the trace process finishes. It may be desirable to extend the contents of an activation point to fit the kind of analysis required on a trace log. Figure 4 shows the fields of an ActivationPoint.

< compiledCode, pc, methodClass, receiverClass,
selector, objectId, action>
compiledCode = thread id
pc = byte code index
methodClass = the class implementing the method
receiverClass = the receiver's actual class
selector = the method name associated with the compiled code
objectId = the hash of the receiver object as an object identity.
action = list of activities associated with the activation point.

Fig. 4. ActivationPoint Tuple

Tracer and TracingService (Figure 5) are two major components of the tracing tool. A Tracer creates and drives the simulated execution of code using its tracing service and keeps the log of activation points. The tracing service performs the actual simulated execution of code and can be derived from existing debugging service or debugging API in the development environment. For multi-threaded programs, each thread is served by one tracer. It also notifies the tracer object when each stepping activity is completed. Additional types of stepping activities are added to the tracing service to address a variety of program behaviors outside the capability of a common debugger. In particular, whenever the program under tracing creates a new thread, the current tracer forks a new tracer instance to trace that thread separately. When that thread finishes, the trace log is merged back to its parent's trace log at the forking point. This flexible approach to tracing multi-threaded programs allows us to obtain trace logs for multiple threads independently. Different trace logs can be combined later for analytical purposes. In latter cases, some preprocessing may be required before trace logs are suitable

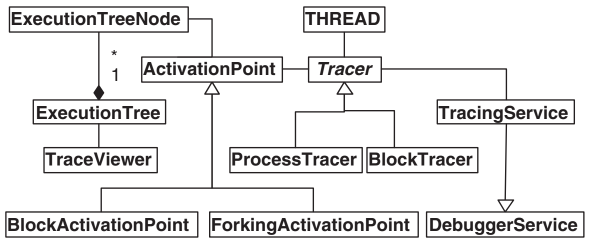


Fig. 5. Class Diagram of the core classes of the Tracer

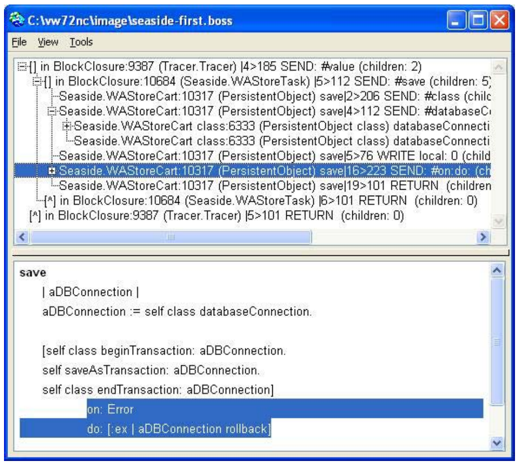
for analysis.

We discovered that a debugging service in VisualWorks always forks service threads to perform its duties. When the tracer is used to trace multi-threading programs, threads under tracing will be scheduled together with those service threads by the system default process scheduler. This may lead to incorrect behavior e.g. the trace log may represent a sequence of events that is different from the true event sequence when the multi-threading program runs outside the tracer. Interesting, this is exactly an instance of the multi-threading programming issue discussed in this paper, when the chain of execution is broken by unexpected collaboration of reusable source codes. In VisualWorks, the DebuggerService is more than just a class library, and it assumes its own model of thread handling, which is in conflict with the tracer thread model. We solved this problem by adapting the tracer to the implicit multi-threading nature of the tracing service built upon debugger service. Specially, we introduced a set of semaphores managed by the main tracer (the tracer that is created first) that imposes the correct execution order on program threads scheduled by the default process scheduler.

In applying our tracing tool for detecting data race conditions on framework-based applications, we generally do not have to trace into the framework code. Instead, the pieces of code in focus will be user-developed parts that will interact with framework components and/or get called by the framework itself. This is partly due to the inversion of control characteristics of framework-based applications [11, Chap1-2]. This fact helps to fight against the log size explosion problem and eases the analysis of trace logs later. To achieve this, users of our tracing tool can define proper configurations for the tracer to run. This method is used to obtain the trace logs for our case studies.

A trace log is first generated as a sequence of activation points. Then, based on the relative relations between activation points, a tree-like structure is built that represents the activation points in calling sequence. Every level of a trace log begins and ends with a pair of activation points that correspond to the first message call and the returning call. As mentioned above, several trace logs can be generated independently or concurrently and then combined for later analysis.

There can be many ways to visualize the tracing log. One of the possibilities is implemented in the TraceViewer. TraceViewer displays the tracing log as a tree-like hierarchy plus reference to the source code with the step taken high-



```
[] in BlockClosure:9387 (Tracer.Tracer) |4>185 SEND: #value (children: 2)
  [] in BlockClosure:10684 (Seaside.WAStoreTask) |5>112 SEND: #save (children: 5)
    Seaside.WAStoreCart:10317 (PersistentObject) save|2>206 SEND: #class (chilc
    Seaside.WAStoreCart:10317 (PersistentObject) save|4>112 SEND: #databaseC
      Seaside.WAStoreCart class:6333 (PersistentObject class) databaseConnecti
      Seaside.WAStoreCart class:6333 (PersistentObject class) databaseConnecti
    Seaside.WAStoreCart:10317 (PersistentObject) save|5>76 WRITE local: 0 (child
    Seaside.WAStoreCart:10317 (PersistentObject) save|16>223 SEND: #on:do: (ch
    Seaside.WAStoreCart:10317 (PersistentObject) save|19>101 RETURN  (children
  [^] in BlockClosure:10684 (Seaside.WAStoreTask) |6>101 RETURN  (children: 0)
[^] in BlockClosure:9387 (Tracer.Tracer) |5>101 RETURN  (children: 0)
```

```
save
    | aDBConnection |
    aDBConnection := self class databaseConnection.

    [self class beginTransaction: aDBConnection.
    self saveAsTransaction: aDBConnection.
    self class endTransaction: aDBConnection]
            on: Error
            do: [:ex | aDBConnection rollback]
```

Fig. 6. Partial view of a trace log inside the TraceViewer

lighted. One can easily trace through the code by navigating the tree hierarchy. A screenshot of the trace log obtained in the Seaside-PPL case study described in subsection IV-B is displayed in figure 6.

Our current implementation of the tracing tool in VisualWorks Smalltalk is depicted by Figure 5. Class ActivationPoint represents an activation point. Its subclasses BlockActivationPoint and ForkingActivationPoint describe an activation point that occurs inside a block or when a child thread forks, subsequently. Tracer class is responsible for most of operations that a tracer requires. BlockTracer and ProcessTracer are used to trace a block of code and a whole process. TracingService inherits from existing DebuggerService in VisualWorks system library and serves the request from tracer objects. ExecutionLog represents the tree-like structure of trace logs and TraceViewer is a GUI tool to visualize the trace log.

B. Data Race Finder (DRF)

The Data Race Finder (DRF) has a simple class model (Figure 7) based primarily in two classes: TracePlayer and TraceCoordinator.

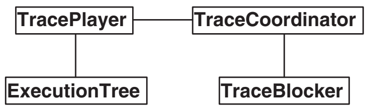
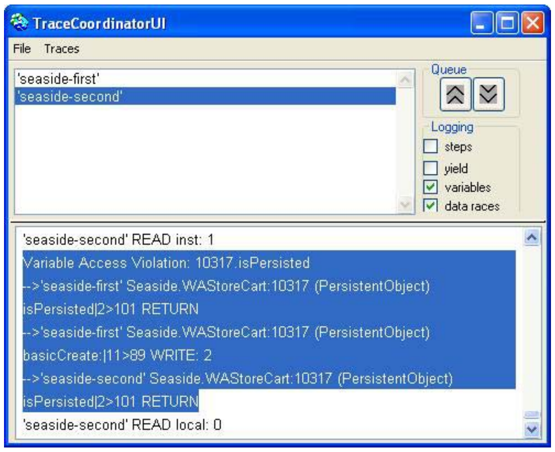


Fig. 7. Classes implementing the Data Race Finder

Instances of class TracePlayer are always associated with one ExecutionTree and one instance of TraceCoordinator. A TracePlayer is able to traverse its ExecutionTree simulating the execution. It means that it recognizes `control primitive` operations. Any scheduling decision is implemented in the class TraceCoordinator. One instance of TraceCoordinator has one association with at least one TracePlayer which informs it about any scheduling related event. Each TraceCoordinator has a queue of ready-to-run players and a collection of objects representing semaphores in use (TraceBlocker). Every running instance of DRF has one TraceCoordinator for scheduling the TracePlayers. TraceBlocker class represents objects that can block the execution of a TracePlayer. Instances of this class are used to represent semaphores and locks. Each instance has a collection of blocked Players.

Users of the tool need to load the Execution Trees that will be part of the simulation. The top pane in Figure 8 is a list representing the ready queue of the TraceCoordinator. As the simulation progresses this list changes because TracePlayer can fork new TracePlayers, finish or get blocked by a blocker.



```
'seaside-first'
'seaside-second'
```

☐ steps
☐ yield
☑ variables
☑ data races

```
'seaside-second' READ inst: 1
Variable Access Violation: 10317.isPersisted
-->'seaside-first' Seaside.WAStoreCart:10317 (PersistentObject)
isPersisted|2>101 RETURN
-->'seaside-first' Seaside.WAStoreCart:10317 (PersistentObject)
basicCreate:|11>89 WRITE: 2
-->'seaside-second' Seaside.WAStoreCart:10317 (PersistentObject)
isPersisted|2>101 RETURN
'seaside-second' READ local: 0
```

Fig. 8. Interface of the Data Race Finder

The bottom area of the tool is the log where messages are printed. The buttons on the ”logging” box control how DRF behaves ones users gives the run command. The options are:

- **steps**: logging each function call in the simulation as seeing by the corresponding TracePlayer
- **yield**: simulating thread scheduling
- **variables**: logging accesses to variables
- **variables →data races** : logging conflicting accesses

IV. Case Studies

We need to test our ideas and techniques on tracing and data race detection. One thing is to validate our claim that by tracing the framework flow of control we can minimize the amount of collected data. Among others, we want to convince ourselves on the effectiveness of post-mortem trace log analysis in data race detection.

We applied the techniques and tools described in previous sections in two following case studies. The first case study is a simple servlet-based web-application where the servlet engine generates concurrent accesses to shared objects in memory. The second case study is a more sophisticated example that uses a web application framework called Seaside together with PPL, an object-relational persistence framework. The case studies illustrate common problems in multi-threading programming with web and server applications as mentioned in the introduction section.

A. Servlet-Based Web-Application

Servlet is a common way to provide dynamic contents for web applications. Servlets work closely with domain-specific frameworks to fulfill the needs of an application. In this case study we build a servlet-based web application that allows users to browse and register themselves for training courses. The servlet engine is provided by Web Toolkit extension in VisualWorks. The example is an extension to Web Toolkit that is part of the Cincom VisualWorks Smalltalk [1]. In this example, Toyz is the name of a fictitious company that offers its own employees to register for on-line courses. To keep the case study simple, domain objects are created and manipulated entirely in the web application memory space. They are shared between various sessions. In reality, however, domain objects are often kept under control by external services such as web services or persistence frameworks, many of them utilize an asynchronous service model.

The servlet ServletAddCourse in Figure 9 is responsible for adding a new course for a registered web user. This servlet inherits from HttpServlet, which allows only a single servlet object to serve many different requests. As usual, the servlet carries out its duty in a http POST request handler, as shown below. However, ServletAddCourse delegates the actual domain processing to a set of utility classes, including a factory class (Toyz). Toyz on its side queries and updates the domain objects to serve each servlet request. In particular, a shared SortedCollection object is used to keep registered courses for all employees. This object is read during the call to getNumberOfRegisteredCoursesFor: and modified in addXref:.

```
ServletAddCourse>>doPost: aRequest response: aResponse
| courseNumber mySession eNumber |
courseNumber := aRequest anyFormValueAt: 'courseNumber'.
mySession := aRequest session.
eNumber := (mySession at: 'signon') number.
Toyz instance addCourse: courseNumber to: eNumber.
aResponse redirectTo: 'j2eeEmployeecourses6.ssp'

Toyz>>addCourse: courseNumber to: employeeNumber
| eName cName num |
num := self getNumberOfRegisteredCoursesFor: employeeNumber.
num < 4 ifTrue:
[eName := self getEmployeeName: employeeNumber.
cName := self getCourseName: courseNumber.
self addXref: (Xref new employeeNumber: employeeNumber
employeeName: eName courseNumber: courseNumber courseName: cName)]
```

Fig. 9. Smalltalk Source Code for Adding Course

We obtain trace logs for servlet execution during http POST requests for two concurrent sessions. The tracer is fine-tuned to trace instructions involving querying and updating list data structures, which are skipped by default. This suggests that while looking for possible data races, we already aim at some suspicious pieces of code. This appears to be productive in practice but requires some a priori knowledge of the code under tracing. The data race detector reveals a data race condition deeply buried inside list data structure operations.

Following the steps of our approach, we traced two concurrent accesses of the servlet and loaded the obtained execution trees into DRF. That sets the scenario when the application servlet is handling two requests for the same domain object. We ran the LockSet algorithm and found a number of conflicting accesses. One of them is presented in Figure 10. We ran the tool again with Green LockSet algorithm and the above conflicting access was not reported.

```
Variable Access Violation: 7250.firstIndex
'servlet-first' SortedCollection:7250 (OrderedCollection) insert:before: WRITE: 1
'servlet-second' SortedCollection:7250 (OrderedCollection) do: LOOP
```

Fig. 10. Output showing a Conflicting Access with Servlets

The first line on Figure 10 is a write operation on the instance variable firstIndex for a SortedCollection object, whereas the second line corresponds to read operations inside a LOOP based on that instance variable.

B. Seaside and PPL

Seaside [2], [10] is a mature Smalltalk framework for building industrial-strength web applications. Seaside framework users can enjoy the power of object-oriented programming in Smalltalk and a rich set of features:

- the servlet architecture of Seaside as a web application server.
- a session object accessible from any web page.
- components (Smalltalk objects) that know how to render themselves as HTML pages and process user requests
- continuation-based multiple control flow that allow developers to approach dynamic web pages in a manner similar to common GUI interface manipulation
- transactions that help isolate a set of related actions (including user interaction) with flow control over page navigation.

We applied our tracing tool to study the architecture of Seaside. In particular, the tracing tool has been used to detect possible racing conditions in web applications built upon Seaside.

Seaside uses the concept of transactions to ensure the correct behavior of flow control. Whenever the user crosses the boundary of a transaction, it is impossible go back to change anything in the state of the program. Actually, a page expired event will be raised and the user is redirected to the current active web page. However, a close look at the transactional mechanism and the continuation-based flow control in general yields that transactional behavior can't be guaranteed with implicit multi-threading programming involvement. Specifically, users can start simultaneously two sessions of the same web application. It is quite often that these two web sessions can share some domain objects. Processing of those shared objects can be delegated to other framework, which may lead to possible data race conditions.

We modify the Sushi web shop example built in Seaside [10] to illustrate how the data race conditions can be observed. The Seaside framework eventually calls the handler WAStoreTask>>go to display a set of dynamic web pages. First, it creates cart, an instance of the shopping cart, Figure 11. This object is shared between working sessions with the same login name. The cart contents are filled in a couple of pages that come next. Finally, the cart is saved using PPL, a persistent framework. PPL was developed in the Illinois Department of Public Health and it was used in several projects [29]. PPL interacts with the underlying database management system using a multithreaded ODBC library. It is that kind of interaction between concurrent accesses initiated by Seaside and PPL, and between PPL and multihtreaded ODBC that raises a data race occurring inside PPL framework.

```
WAStoreTask >> go
| shipping billing creditCard |
person := Person new.
self isolate: [[self login] whileFalse].
WACartFactory instance connect.
"cart instance is shared between sessions"
cart := WACartFactory instance retrieveCart: person name.
self isolate:
[[self fillCart.
self confirmContentsOfCart] whileFalse].

self isolate:
["save the cart using a multithreaded persistence framework"
cart save.
self displayConfirmation]
```

Fig. 11. Implementation of Operation go in Seaside

PPL assumes that it is in control of its internal variable isPersisted. However, that assumption is undermined when the asynchronous ODBC call yields its thread of execution. Figure 12 shows the implementation of operation saveAsTransaction on PPL.

```
PersistentObject>>saveAsTransaction: aDBConnection
self isPersisted
ifTrue: [self update: aDBConnection]
ifFalse: [self create: aDBConnection].
self makeClean
```

Fig. 12. Implementation of saveAsTransaction in PPL

We obtain the trace logs for cart save during two executions of the above code in a Seaside application. Note that the line of code cart save is protected inside the block `isolated:`. Seaside uses such blocks to ensure transactional behavior among a sequence of web page accesses. The data race detection tool reveals possible data race conditions between these two executions. In particular, it reports concurrent accesses with at least one write to instance variables of the cart object as suspicious places for data races (Figure 13).

```
Variable Access Violation: 10317.isPersisted
'session-first' Seaside.WAStoreCart:10317 (PersistentObject) isPersisted RETURN
'session-first' Seaside.WAStoreCart:10317 (PersistentObject) basicCreate: WRITE: 2
'servlet-second' Seaside.WAStoreCart:10317 (PersistentObject) isPersisted RETURN
```

Fig. 13. Output showing a Conflicting Access with Seaside And PPL

In the above variable access violation report the instance variable isPersisted of cart object WAStoreCart:10317 is concurrently read by the method **isPersisted** and written by method **basicCreate:** of the same object.

V. Related Work

A. Tracing methodology

Our approach to tracing object-oriented programs relies on debugging mechanism. Actually, the debugger is used as a simulator to interpret program instructions and collect tracing information. Reusing the debugger can be considered just as an implementation advantage, since the main goal is to be able to drive the simulated code and to examine program state after each stepping activity. Several extensions and adaptations are made to existing debugging machinery to meet the demands of the tracing process.

Tracing programs based on debugging technology has been used in other works [14], [18], [25]. Lange and Nakamura built a tool called Program Explorer to trace C++ object-oriented programs. Their approach consists of applying trace points (similar to debug breakpoints) to program source code that can reveal a C++ object's `this` pointer to a trace recorder. Programs are then recompiled and trace events are generated during execution. Lencenvicius et al used a debugger written in Java to continually monitor the results of a user-specified query string while a Java program is running. This customized debugger instrumented Java class files to invoke the debugger at points specified by the dynamic query. Hamou-Lhadj and Lethbridge provided a survey of representative trace exploratory tools and techniques, among which debugger-based tools in Java such as Shimba [27] generally require setting breakpoints at places of interest and then executing target systems under control of a customized debugger.

The strength of our approach consists of the following:

- The tracer interprets program instructions to generate trace logs containing required information for analytical purposes.
- The tracing process is carried out with unmodified program source code, without any prior program instrumentation and trace-points setting. We need only to specify the block of code that requires tracing and an optional tracing configuration.
- In general, we don't need to trace the whole application. In contrast, most of the times we can limit the tracing to the flow of control of a framework. It does not require deep/exhaustive knowledge of the traced program, allowing exploratory understanding of unfamiliar code.
- The tracer is specially designed and fine-tuned to trace multi-threading programs.

Our tracing methodology differs from the above-mentioned approaches in that it interprets the code under tracing to gather the tracing information selectively. Such an approach allows to generate extra tracing information post-mortem, e.g. to process the trace log and insert additional tracing information regarding accesses to method parameters. On the other hand, this approach can also have some disadvantages. Since the method invocation context is examined on every step, performance may suffer. However, the interactive usage of the tracer to trace small to average block of code makes this disadvantage not a very serious obstacle. On the other hand, flexible configurations allow us to obtain trace logs quickly for the case studies in section IV.

B. Analyzing Trace Logs

Burrows et al [26] describe a tool called Eraser that is capable of detecting data races in lock-based multi-threading programs using an efficient implementation of the LockSet algorithm. Eraser instruments a binary program to generate the trace log for data race detection and is generally not limited to object-oriented programs.

Lange and Nakamura [17] suggest efficient techniques of merging and pruning objects and method calls to reduce the trace log search space. Much effort has been spent on visualization and statistical discovery to extract useful information from program traces [14], [19]. Jinsight [22] is a Java visualization program that can display a program execution view and help comprehend the behavior of multiple threads in Java programs. Various filters [18] can be applied to trace logs to get simpler representations. However, low-level filters (filters on function calls, for example) have a disadvantage that they can not show a good interaction between remaining elements, and the information contained in filtered out elements are lost. It is important to know what to filter, where to filter and how to represent what remains after filtering. Recently program query languages have been proposed over traces to study dynamic program behaviors [9], [13], [20]

To approach the problem of finding race conditions from program dynamic information we propose the concept of merging different trace logs (each for one thread of execution). Each program thread can be traced separately (using a tracer object in our design) and then merged back when the trace log is ready. A program analyzer will visit the nodes inside the trace logs and perform required analytical activities. Trace log information is kept as documentation of the frameworks, this is different from [7] and [28]. Every trace log node holds information regarded the identity of the object receiving the message call, a pointer to its source code, and the characteristics of the message call. Details on how the data races are detected using the combined trace log is discussed in subsection III-B.

Instead of detecting possible race conditions in existing programs, Boyapati et al [6] suggest a completely different approach to prevent data races and deadlocks in the first place, using a static type system extension to the Java source language. A corresponding type system extension for the Java virtual machine language is built, which results in a language called SafeJVML. Well typed SafeJVML programs are shown to be free of data races and deadlocks [5].

VI. Conclusion

Multi-threading programming offers a trade-off between responsiveness and complexity. Multi-threading applications have better responsiveness but they are more complex to develop and maintain. Frameworks are also affected. Frameworks seem to hide the complexity of multi-threading by controlling the creation and control of threads, but if application domain objects are shared by two different threads then there is the possibility of data races. Because frameworks hide the creation and control of threads, it can be hard for application programmers to find the data races, even though the data races are in the code that they wrote.

Since framework-based applications and multi-threading programming are common, developers may need techniques and tools that can keep them aware of such pitfalls and avoid unwanted outcomes. Test cases may help in first place. However, data races can be present in a program even if the results in a external module such a database appear to be correct during some testing execution. Changes to external libraries or components of a system can make data races to surface. Having a better way to detect possible data races will be of great help to increase software quality. We presented a technique that uses tracing to find conflicting accesses. We extended the LockSet algorithm with green thread semantics. Conflicting accesses are good approximations to data races. We found that Execution Trees are good documentation of the flow of control of a frameworks and that they can be used effectively in finding data races. Execution Trees can also be used as an exploration tool when considering changes in the framework or calls to external libraries.

The design of the tools and data structures presented in this paper are guidelines to others who want to implement similar utilities in other languages. They will be specially useful in reflective languages that provide access to bytecodes and other artifacts that help simulate the execution of virtual machine.

As we showed in the paper, the structure of frameworks can be used to reduce the amount of tracing so that programmers do not get overwhelmed with tracing data. Moreover, the fact that the internal details of frameworks that do not interest them can be hidden alleviates the need to process large volume of trace logs and thus helps application developers focus on their own code. We applied our technique and tools to case studies with servlets and web applications based on the Seaside framework. The tool reveals a few data race conditions and confirms our above arguments on data races on framework-based applications.

Acknowledgment

The authors would like to thank Stéphane Ducasse, members of the Software Architecture Group (SAG) at University of Illinois at Urbana-Champaign and the anonymous reviewers for their valuable comments to the paper.

References

[1] Cincom smalltalk. http://smalltalk.cincom.com.

[2] Seaside. http://www.seaside.st.

[3] Lucy Berlin. When objects collide: Experiences with reusing multiple class hierarchies. In *Conference on Object Oriented Programming, System, Languages and Applications*, pages 181–193, 1990.

[4] Joshua Bloch and Neal Gafter. *Java(TM) Puzzlers : Traps, Pitfalls, and Corner Cases*. Addison-Wesley, 2005.

[5] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, 2004.

[6] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. A type system for preventing data races and deadlocks in java programs. In *Conference on Object-Oriented Programming Systems, Languages and Application*, pages 211–230. ACM Press, 2002.

[7] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Conference on Programming Language Design and Implementation*, pages 258 – 269. ACM Press, 2002.

[8] Desmond D'Souza, Aamod Sane, and Alan Birchenough. First-class extensibility for UML packaging of profiles, stereotypes, patterns, 1999.

[9] Stéphane Ducasse, Michael Freidig, and Roel Wuyts. Logic and trace-based object-oriented application testing. In *International Workshop on Object-Oriented Reeingeering*, 2004.

[10] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside - a multiple control flow web application framework. In *In Proceedings of European Smalltalk User Group Research Track*, 2004.

[11] Mohamed Fayad, Doug Schimidt, and Ralph Johnson, editors. *Object-oriented Foundations of Framework Design*. Willey, 1999.

[12] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. In *IEEE Software*, 1995.

[13] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Application*, pages 385–402. ACM Press, 2005.

[14] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, 2004.

[15] Paul Hyde. *Java Thread Programming: The Authoritative Solution*. Sams, 2000.

[16] Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[17] Danny Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Conference on Object-Oriented Programming Systems, Languages and Application*, pages 342–357. ACM Press, 1995.

[18] Danny Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, 30(5):63–70, 1997.

[19] Darko Marinov and Robert O'Callahan. Object equality profiling. In *Conference on Object-Oriented Programming Systems, Languages and Application*, pages 313–325. ACM Press, 2003.

[20] Michael Martin, Benjamin Livshits, and Monica Lam. Finding application errors using PQL: A program query language. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Application*. ACM Press, 2005.

[21] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[22] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the execution of java programs. In *Seminar on Software Visualization*, 2002.

[23] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems*, 1998.

[24] Monica Pawlan. Multithreaded Swing applications. Sun Developer Network. http://java.sun.com/developer/technicalArticles/Threads/swing/, September 2001.

[25] Ambuj Singh Raimondas Lencevicius, Urs Hoelzle. Dynamic query-based debugging. In *European Conference on Object-Oriented Programming*, pages 135–160. Springer Verlag, 1999.

[26] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 1997.

[27] Hausi Müller Tarja Systä, Kai Koskimies. Shimba - an environment for reverse engineering java software systems. *Software Practice and Experience*, 31(4):371–394, 2001.

[28] Christoph von Praun and Thomas Gross. Object race detection. In *Conference on Object-Oriented Programming Systems, Languages and Application*, pages 70–82. ACM Press, 2001.

[29] Joe Yoder, Ralph Johnson, and Quince Wilson. Connecting business objects to relational databases. In *Fifth Conference on Patterns Languages of Programs*, 1998.