Enabling Modularity and Re-use in Dynamic Program Analysis Tools for the Java Virtual Machine

Danilo Ansaloni[1], Stephen Kell[1], Yudi Zheng[1], Lubomír Bulej[2], Walter Binder[1], and Petr Tůma[2]

[1] University of Lugano, Switzerland
firstname.lastname@usi.ch
[2] Charles University, Prague, Czech Republic
firstname.lastname@d3s.mff.cuni.cz

Abstract. Dynamic program analysis tools based on code instrumentation serve many important software engineering tasks such as profiling, debugging, testing, program comprehension, and reverse engineering. Unfortunately, constructing new analysis tools is unduly difficult, because existing frameworks offer little or no support to the programmer beyond the incidental task of instrumentation. We observe that existing dynamic analysis tools re-address recurring requirements in their essential task: maintaining state which captures some property of the analysed program. This paper presents a general architecture for dynamic program analysis tools which treats the maintenance of analysis state in a modular fashion, consisting of *mappers* decomposing input events spatially, and *updaters* aggregating them over time. We show that this architecture captures the requirements of a wide variety of existing analysis tools.

1 Introduction

To gain insight about how to optimise, debug, extend and refactor large systems, programmers often rely on dynamic program analysis tools, which observe a program in execution and report properties of that execution. Many popular bug-finding and profiling tools are of this form, including the Valgrind suite [25], DTrace [5], VisualVM[3], GProf [13] and others, while research literature continues to propose diverse new tools for data race detection [11], white-box testing [29], security policy enforcement [10] and more.

Implementing such tools is unduly difficult. One recurring source of difficulty is that high-level requirements must be translated into code reacting to low-level execution events. For example, a simple context-sensitive memory profiler, which must "count allocated bytes, totalled by call chain" must be written in terms of method entries and exits, a variety of low-level object allocation mechanisms, and so on. Although several existing frameworks assist with creation of dynamic analyses, all support only the same broad approach, which we characterize as

[3] http://visualvm.java.net/

control flow interposition: they provide the interception mechanisms for various control-flow events, but it remains the user's responsibility to describe how their analysis abstracts and aggregates these events. This approach is found in bytecode transformation libraries such as ASM[4], Soot[5] or Javassist [6], in aspect languages such as AspectJ [18], in external domain-specific languages (DSLs) such as DTrace's D [5][6] and in embedded DSLs such as BTrace[7] or DiSL [23]. By abstracting only the control-flow aspect of the task, considerable work is repeated by successive tool authors in bridging the abstraction gap between high-level tool requirements and low-level instrumentation.

In this paper we present a system for describing dynamic analyses more succinctly using a contrasting approach which we describe as *state-oriented*. The analysis' requirements are decomposed in terms of the structures which hold the accumulated state of the analysis, and the semantics with which these structures evolve. For example, our allocation profiler is defined as a composition of a call chain recorder, a map from call chains to counts, an encapsulated definition of allocation events, and an updater function which increments the relevant count on each allocation. All these are re-usable library components. In our system, dynamic analyses are built straightforwardly by using short script-style code fragments to combine generic data structures and state transformers.

Our goal is that using such a system, a large proportion of analysis requirements can be catered for almost entirely by library code. In other cases, new requirements can easily be satisfied by creating small extensions of existing library code. In rare cases where library code cannot be combined or extended to satisfy requirements directly, the full expressiveness of control-flow interposition is available, since our system builds on an existing, more conventional framework. So, while we preserve the expressiveness of existing control-oriented systems, common cases are handled using library code rather than new user code.

The contributions of this paper are threefold.

- We describe the design of a framework for composition of dynamic analyses, focusing on its API and three key constituent interfaces which facilitate our state-oriented decomposition: *instrumentations* emitting events when relevant code regions are being executed; *updaters* describing how the analysis's state (meaning the state which is used to produce the analysis's output, such as profile data or execution monitor state) responds to new events; and *mappers* routing events to the subset of output state requiring consequent update.
- We show with examples that this factoring can express a wide variety of analyses, including substantial real use cases presented as case studies.
- We show experimentally that our system offers performance generally competitive with lower-level frameworks used like-for-like.

We begin by motivating the high-level design of our system.

[4] http://asm.ow2.org/
[5] http://www.sable.mcgill.ca/soot
[6] We note that control-flow abstractions are a characteristic of common DTrace providers, but not necessarily of the whole framework.
[7] http://kenai.com/projects/btrace

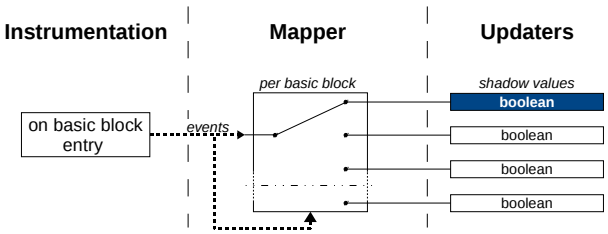
2 Motivation

In this section we motivate our work by showing that (1) significant latent commonality exists among dynamic analysis tools, (2) frameworks currently used to build them do not extract this commonality, and (3) a more state-oriented decomposition of analyses can ameliorate this. We consider these issues in turn.

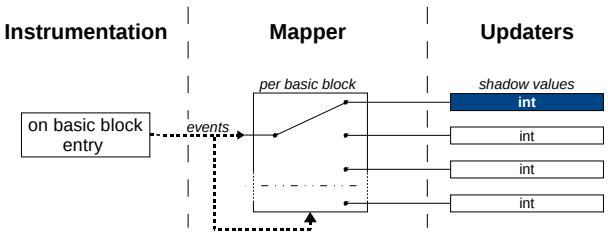
2.1 Latent Commonality

Instrumentation is only one of the recurring sources of complexity in dynamic analysis tools. The state maintained by dynamic analysis tools is commonly explained using the concept of *shadowing*: the job of the analysis is to update a set of shadow values that correspond to program state elements such as objects, fields or threads. The shadows' role is to maintain additional data about the program's execution so far, supplementing the program state. Each shadow value is updated in response to incoming events. Update rules might simply involve incrementing an integer (in the case of counters maintained by a profiler), or might manipulate a set (in the case of a data-race detector based on lock sets), propagate taint bits (in the case of an information flow analysis), advance a finite state automaton, and so on. Our insight is that by separating out the programmer's description of *how* shadows are represented and updated from *what* unit of program state is modelled by each shadow, greater commonality can be extracted re-usably from distinct tools.

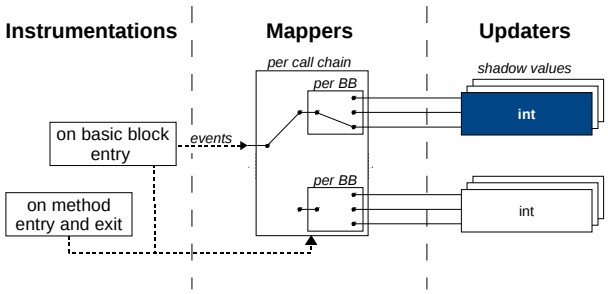
To illustrate this, we observe how a variety of useful tools can be constructed by independently recombining different shadow value mappings (i.e., what elements of program state to shadow) with different update rules (i.e., how to shadow them). First, consider a simple code coverage tool working at the basic block level. It consists of (1) an instrumentation of basic block entries, (2) a mapper associating a shadow boolean value to every distinct basic block ID, and (3) an updater that, for each basic block ID that is received, sets its shadow boolean to true.



Instead of coverage, suppose we now require a count-based profiling tool. We shadow the same program state elements, but now with an integer updated by increments (instead of a boolean updated by set-to-true).



Now consider a context-sensitive profiler. We keep the same updater, but maintain each shadow per call chain. This means our mapper now has two levels: from call chain, to basic block, to the counter payload. The set of call chains must itself be constructed by additional instrumentation, applied to method entry and exit, typically to maintain a calling context tree [1].



Note that the overall form of the system is still the same, and it contains the same kinds of units: an instrumentation that observes events of interest, a mapper of such events to the relevant part of the analysis state (possibly over multiple stages), and updaters of individual state elements in response to events to reflect the context information available for such events. An interesting property is that the mapping logic can itself maintain state and be sensitive to events gathered using instrumentation, as with the call chain in the latter example.

To move from each example to the next in the above series, we change either how each analysis state element (shadow) is represented and updated (booleans updated by set-to-true, versus integers updated by increment), or what spatial structure of shadow elements the gathered events are mapped onto, including how fine-grained this is (from "one shadow per basic block" to "one per basic block, per call chain"). However, in no case we change both at the same time. Therefore, if it were possible for the analysis developer to specify all these concerns independently, each analysis could be constructed very simply by re-using pieces of the previous one. Achieving this independence is a key contribution of our API design (§3.4).

2.2 Limitations of Current Infrastructure

Fig. 1 summarises our goals: to provide still greater ease of use without sacrificing flexibility. To achieve ease of use, we seek greater re-use in the code used to

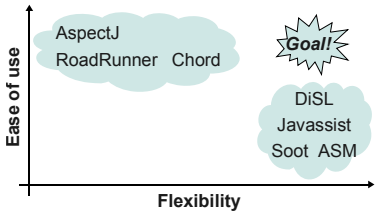


Fig. 1. Trades of ease-of-use with flexibility in existing frameworks

construct dynamic analysis tools. Our insight is that despite the latent commonality we saw in the previous section, current infrastructure makes it either impossible or extremely difficult to structure dynamic analysis tools so that this logic can be isolated and re-used. We survey these existing infrastructures in two broad categories: low-level libraries for code transformation, and higher-level instrumentation-based frameworks.

Low-level libraries A basic yet popular infrastructure supporting dynamic analyses is libraries for manipulating code representations such as Java bytecode [20], VEX [25], LLVM [19], or machine instructions. Convenient APIs for transformation of this representation include libVEX[8], ASM[9], Soot and Javassist [6]. These reduce the effort required to perform common-case transformations, relative to manual coding. However, they do not abstract away the complexity of this representation, since the programmer must still have a thorough understanding of it in order to use the API.[10]

Fixed-event-set frameworks Many analysis frameworks focus on relieving the programmer of the instrumentation burden, by abstracting various low-level instrumentations into the form of a fixed set of hooks (to which callbacks may be registered) or join points (for which advice may be provided) or events (for which handlers may be defined). These frameworks are valuable, but their API designs limit their ability to construct analyses by composition, as we now detail.

Unicast events In Chord[11], for example, an analysis is defined as an implementation of the EventHandler abstract class, implementing a fixed set of callback-style methods. Since there may be only one EventHandler instance deployed by a given analysis, there is no convenient way to combine independent analyses reacting to the same or overlapping sets of events (without manual forwarding code). The pipeline architecture of RoadRunner [12] also

[8] http://www.valgrind.org/
[9] http://asm.ow2.org/
[10] We note that these systems are powerful, and are not limited to the construction of analysis tools; they can replace or transform entire sections of code in arbitrary ways. Here, we are referring only to their suitability with respect to construction of program analysis tools (a common use case).
[11] http://pag.gatech.edu/chord/

suffers from a unicast limitation, in that once an event has been discarded by an earlier pipeline stage, it is unavailable to all later stages.

Event forwarding requirements Since Chord targets Java, where only single inheritance is provided, it cannot combine independently-written handlers for disjoint sets of events without a specially crafted EventHandler that forwards events as required.

Limitations of pipeline-style composition Although RoadRunner provides an apparently modular pipeline-based composition model, each pipeline stage makes strong assumptions about the shadowing and forwarding behaviour of the rest of the pipeline.[12] As a result, although pipeline stages are replaceable by algorithmically distinct alternatives (a primary design goal of the framework [11]) they are not highly re-usable, since each one is usable only in a pipeline contexts similar to the one for which it was designed.

Fragile base classes In both Chord and RoadRunner, defining a new kind of event creates fragile base class problems: in Chord, it requires defining a new Instrumentation class (using the lower-level Javassist library) dispatching to a new EventHandler-like base class. Existing analyses cannot be rebased onto this new superclass without forwarding code. RoadRunner has a similar property: since events are propagated through distinct methods in the Tool base class, defining new events means modifying this base class.

To avoid the problems just seen, we must design an API that does not model event transmission as delegation within an inheritance hierarchy, nor as method calls across a fixed interface.

2.3 State-oriented Abstractions

Besides an appropriate event abstraction, we must provide additional abstractions to capture the commonality seen in §2.1. Invariably, as we saw with the examples of §2.1, non-trivial dynamic analyses maintain some kind of state; describing this state can offer a succinct way to describe the analysis.

We claim that any instrumentation-based dynamic analysis may be decomposed roughly using the following equation.

$$\textit{Analysis} = \textit{Instrumentations} + \textit{Mappers} + \textit{Updaters}$$

Instrumentations provide events from the base program. *Updaters* modify the "primary" state of the analysis, meaning the information whose collection is the end goal of the analysis. This might include the relative hotness of each method or basic block (for a CPU profiler) or the state of each active execution monitor (for a bug-finding analysis). Updaters include the logic which updates these values in response to new events.[13] Finally, *mappers* connect the other

[12] We note that RoadRunner's implementation of shadow state allows each field in an object to be shadowed, but by at most one pipeline stage at a given time. This forces pipeline stages to coordinate hand-off, using a shared state machine.

[13] We will sometimes prefer to talk about collective "analysis state", and sometimes about many individual "shadow values".

two layers; logically speaking, they route the incoming events to a subset of the relevant shadow values.

A key insight of our design is that the mapping stage is often an independently re-usable part, or, in complex cases, a composition of such parts. For example, in §2.1 we saw context-sensitive data collection implemented using a calling context tree. Such a tree represents an intermediate mapping layer which is independently re-usable across many analyses. In general, the same update logic might be valid for a variety of granularities of analysis—depending, for example, on whether profiles are being collected per method or per call chain, whether monitoring is done per-object or per-class, and so on. A separate mapping stage allows these granularities to be defined independently from other aspects of the analysis.

Table 1 offers some credence to the claim of our equation's generality. It shows several analyses, including both simple examples of the kind seen earlier (§2.1) and complex examples from the literature, decomposed according to this equation. The table columns refer to the concepts we have by now introduced. The "cardinality" column elaborates on the role of the mapping stage: most mapping stages are designed to maintain one shadow value per some division of program state (current or former), such as "one state machine per live object" or "one count per call chain seen", and we detail this explicitly.

To realise this decomposition, we require a programmatic way to express each of the three parts separately. Furthermore, our programmatic realisation must be efficient so as to meet the requirements of dynamic program analysis. In the next section, we therefore describe this programmatic realisation in the form of an API design, paying special attention to both the overall ease-of-use and the essential implementation details on which the design's practicability depends.

3 API Design

In this section we present FRANC, our FRamework for ANalysis Composition, which is implemented on top of the DiSL instrumentation framework[14] [23], and which models our state-oriented decomposition of dynamic analysis tools. Like DiSL, FRANC targets Java bytecode. We chose DiSL partly because it is an open-source framework and (to our knowledge) the only one providing full bytecode coverage out-of-the-box. Whereas DiSL provides instrumentation primitives and a set of annotations allowing Java-language code snippets (static methods) to be inlined in a user-specified manner, FRANC instead focuses on encapsulating instrumentation behind "listener"-like[15] event interfaces, and subsequently processing events in plain, annotation-free Java code. Instrumentations are event producers. The remainder of the analysis is implemented as event consumers which subscribe to some set of instrumentations. The underlying DiSL API is available to handle rare cases where library-defined instrumentations do not suffice. In overview, our API provides the following features:

[14] http://disl.ow2.org/

[15] We use "listener" in the same sense as various Java libraries, notably Swing [33].

name	instrumentations	mappers		updaters	
		function of	cardinality	shadow values	update rule
basic block coverage	basic block entry	event only	per basic block	boolean	set true on entry
basic block hotness	basic block entry	event only	per basic block	counter	increment on entry
context-sensitive basic block profiler	basic block entry	calling context tree	per call chain × basic block	counter	increment on entry
context-sensitive allocation profiler	memory allocation	calling context tree	per allocation call chain	counter	increment (by alloc size) on entry
cache simulator (e.g. [32])	memory read/write	accessed line + cache set state	per access site × $\{hit, miss\}$	counter	increment on hit/miss
ElephantTracks [26]	method entry/exit object allocation object use object reclamation reference update	event + timestamp (using logical clock)	per object + single value (logical clock)	object last-use time referenced objects	Merlin algorithm [14]
listener latency profiling [16]	method entry/exit	event + timestamp (using system clock)	per subtype of EventListener	($Count$, $AverageDuration$)	running average update
typestate checker [31]	method entry	event only	per live object	typestate automaton	advance, else report violation
concolic execution [29]	value-generating computation, branches	path constraint	per program value	SMT formulae	update + simplify
Eraser race detector [28]	lock ops, field read/write	event only	per pair ($object$, $field$)	consistent lock set	initialize, then set-intersect; report if empty
dynamic shape analysis [17]	garbage collector events: object scan complete, heap traversal complete	event only	per class	class-field summary graph	initialize, then invariance check
DefUse [30]	field read/write	event only	per field	writing thread ID or instruction	invariance check

Table 1. A series of distinct dynamic analyses decomposed into events, mappers and shadow values

– library-defined event sources, implemented by bytecode instrumentation;
– mappers that aggregate, decompose, augment and/or filter incoming events—and may be built compositionally from other mappers;
– updaters that comprise a representation for an element of analysis output state and logic for updating it in response to events;
– "auto-completion" features for inferring common wiring patterns between event producers and consumers.

Clients of the FRANC API are submitted to a weaver which analyses the client program's use of various API interfaces and performs the necessary instrumentation. Our use of a Java API reflects our implementation, which is both a Java-based API and performs instrumentation of Java bytecode. However, we believe the same conceptual decomposition could be implemented conveniently in many other instrumentation scenarios.

3.1 API Elements

Each element of our equation (§2.3) corresponds to a small set of API-provided definitions. We briefly survey these now, before proceeding to a full example.

Instrumentations We can construct instrumentations using code like the following. FieldAccess is one of a collection of *code region* marker interfaces identifying various bytecode patterns which can be instrumented.

```
Instrumentation<FieldAccess> faInstr = new Instrumentation<>();
```

Scheduling interfaces Code which consumes instrumentation events implements another kind of marker interface, called scheduling interfaces, that decide whether the code executes *Before* or *After* the instrumented feature (among other options). Here we begin a class that will define code to run after each field access.

```
class FieldMapper
  implements After<FieldAccess>
  // ... to be continued
```

Shadow value maps The shadow state of our analysis is usually stored in some object implementing Java's standard Map interface. Here we store a set of counters, indexed by strings (whose values will be explained shortly).

```
Map<String, AtomicLong> fieldAccesses = new HashMap<>();
```

Mappers Mappers are a particular kind of client of instrumentation. Usually, they use instrumentation events to maintain a current value—on either a global, per-thread, or some other basis. The current value denotes the key (in the shadow state map) that is currently relevant for update. For example, our field access mapper holds a string representing the most recently accessed field. We represent these current values using familiar Java data types (modestly generalised) such as ThreadLocal. We have already seen the beginning of the mapper definition; in full, it is as follows.

```
class FieldMapper
  extends ThreadLocal<String>
  implements After<FieldAccess> {
    public void after(FieldAccess codeRegion)
    { set(FieldAccessContext.getFullFieldName(codeRegion)); } // grab current field
}
FieldMapper currentField = new FieldMapper();
```

Context accessors The helper class FieldAccessContext, which retrieves a unique string identifying the field, belongs to a library of context accessor classes, whose design confers important modularity properties. We return to this in §3.3.

Updaters How does a new FieldAccess event update the relevant element of the fieldAccesses map? This is encapsulated in an updater object. Ours comes from the library, and has simple "increment" semantics.

```
Analysis<FieldAccess> updater = new PostIncrement<>(fieldAccesses, currentField);
```

Subscription relationships To "wire up" the system, we can explicitly create subscription relationships to route events from an Instrumentation to its clients—here both the mapper and the updater.

```
faInstr.appendSubscriber(currentField);
faInstr.appendSubscriber(updater);
```

3.2 Complete Example

We now build a complete example combining the elements we just saw. Fig. 2 shows a simple analysis which counts accesses to fields and groups them a unique identifier for that field.[16] This analysis could be used to profile hotness of fields, as a basis for cache optimisation.

As is evident from the code, a few minor tweaks have been made. These are all changes which, it turns out, help to make the pieces fit together straightforwardly. We discuss each of these in turn.

```
1  // shadow values
2  Map<String, AtomicLong> fieldAccesses = new ShadowMap<>(...);
3
4  // (a) first define a stateful mapper which latches the field being accessed
5  class FieldMapper
6    extends ThreadLocal<String>                 // 1. maintain a String per thread
7    implements AfterCompletion<FieldAccess> {   // 2. by hooking field accesses
8      public void afterCompletion(FieldAccess codeRegion)
9      { set(FieldAccessContext.getFullFieldName(codeRegion)); } // grab current field
10 }
11 // (b) then instantiate it
12 FieldMapper currentField = new FieldMapper();
13
14 // update rule: atomic counter increment, from the library
15 Analysis<FieldAccess> updater = new PostIncrement<>(fieldAccesses, currentField);
16
17 // deployment -- infers what instrumentations are needed
18 FRANC.deploy(FRANC.complete(currentField, updater));
```

Fig. 2. Composing a simple field access counter

[16] This identifier is defined carefully so as to account for like-named fields occurring within the same class (by inheritance), and for distinct class loaders.

```
1 class PostIncrement <T extends CodeRegion> implements AfterCompletion<T> {
2   Map<?, AtomicLong> map; MutableReference current;
3
4   public PostIncrement(Map<?, AtomicLong> m, MutableReference c)
5   { map = m; current = c; }
6
7   public void afterCompletion(T codeRegion)
8   { map.get(current.get()).incrementAndGet(); }
9 };
```

Fig. 3. A value updater included in our library—in this case, an atomic incrementer

Shadow value map Although we could use various Map implementations to hold our analysis's shadow state (line 2 in Fig. 2), we use our own ShadowMap. This is somewhat tailored to analysis use. In particular, ShadowMap additionally allows a factory class to be supplied, enabling get(K key) to create a new map entry in the case of an absent key.

Mapper The mapper (line 5 in Fig. 2) is essentially unchanged from earlier. Since our analysis groups field accesses by the field's unique identifier, it needs to "latch" this identifier whenever a field is accessed. Since each thread may be accessing (at most) one distinct field at a time, our mapper state is a ThreadLocal string value, recording the unique identifier of the last-accessed field. Recall that this state will be used to arrange that each field access is "mapped onto" a different shadow value (i.e. element in the map) depending on what field is being accessed.

Scheduling interfaces One change is that our mapper implements the AfterCompletion<FieldAccess> marker interface, instead of plain After<FieldAccess>, with the effect that it runs after only those field accesses which complete non-exceptionally.

Context accessors It is worth paying closer attention to how the context information is accessed, in line 9 of Fig. 2, using context accessors. These form a separate class hierarchy than both the scheduling interfaces (such as Before, After) and the code region markers (such as FieldAccess). Here, the FieldMapper calls a static method on FieldAccessContext to extract the field's unique identifier. All access to context is done through such methods, and always passing the current code region, which represents the instrumentation site. This allows us to statically define the relationship between context information and the instrumentation sites at which it is available. As a concrete example, the FieldAccessContext class requires that its codeRegion arguments are in fact FieldAccess instances. Consequently, even though context is defined in a separate class hierarchy, programmer errors involving inappropriate context selection are caught statically. By contrast, these errors are only caught dynamically under existing systems (such as DiSL, custom instrumentations in Chord, etc.). By extending this approach to custom dynamic context, to be described in §3.3, we also avoid many scenarios in which the programmer would otherwise resort to accessing the operand stack directly—another source of run-time errors.

Updaters For each field access we intercept, we require an update function for the shadow values stored in the map. This is a simple increment. More precisely, it is a *post-increment*, meaning the increment happens *after* the completed execution of the underlying bytecode (here, after each field access). As shown in Fig. 3, our library defines PostIncrement for this purpose, appropriately parameterised. The user needs only to instantiate it. Note that we pass it a reference to our currentField mapper state, and to the map itself. All that the incrementer knows about the mapper is that it defines a method get() which will supply some object (here the field name) usable as a key to index the map.[17] This "thin waist" hourglass design is what allows many distinct updaters to be composed unmodified with many distinct mappers.

Autocompleted instrumentation Previously (§3.1), we saw how to manually instantiate Instrumentation objects which act as sources of field access events. In practice, in this example we do not need to construct an Instrumentation explicitly, since our updater implements Analysis<FieldAccess>; instead, we construct the analysis with FRANC.complete(), which infers (from types of its arguments) that a source of FieldAccess events must be added to the composition. Since there is only one such source defined in the library, it is implicitly added. In some more advanced cases, instrumentations will need to be explicitly constructed.

Ordering So far, a subtle question has gone unanswered. We have two distinct clients of FieldAccess events, and they share state. In what order does their code run? Our example requires that the FieldMapper runs first, so that when the updater calls get(), it sees an up-to-date unique field identifier. Deployment logic (in FRANC.deploy()) embodies a series of rules for deciding this ordering. By default, these arrange that mapper logic runs before updater logic, though the user may also specify an ordering explicitly. We discuss this issue further in §3.5.

```
1  /* marker superinterface for defining features in bytecode (body, basic block, ...) */
2  public interface CodeRegion {}
3
4  /* superinterface for instrumentation-sensitive code */
5  public interface Analysis<T extends CodeRegion> { /* see Sec. 3.3 */ }
6  /* now, fine-grained interfaces for before/after/...  */
7
8  // runs before bytecode of interest
9  public interface Before          <T extends CodeRegion> extends Analysis<T>
10 { void before(T codeRegion); }
11
12 // run after bytecode of interest completes non-exceptionally
13 public interface AfterCompletion<T extends CodeRegion> extends Analysis<T>
14 { void afterCompletion(T codeRegion); }
15
16 // run after, in exceptional case
17 public interface AfterThrowing <T extends CodeRegion> extends Analysis<T>
18 { void afterThrowing(T codeRegion); }
19
20 // run after, in both exceptional and non-exceptional cases
21 public interface After          <T extends CodeRegion> extends Analysis<T>
22 { void after(T codeRegion); }
```

Fig. 4. Marker "scheduling" interfaces for code invoked at instrumentation sites

[17] Here, get() is defined by ThreadLocal; see §3.4.

3.3 Open, Re-usable Event Definitions

The library defines a large number of specialised CodeRegion marker interfaces such as BasicBlock, FieldAccess, ArrayAccess, Body (meaning method or constructor body), and various others (also including most individual bytecodes). Each represents some feature found in Java bytecode. In all cases, the user can specify code that should run before the feature, after it, and so on, as shown in Fig. 4. A key property of our API design is that the set of events is extensible, and that the information accompanying each event is also (independently) extensible, and that accesses to this information are statically checked. Here we review how this is achieved.

Context information For any event, there will be some context information that is accessible from all occurrences of such an event. For example, the context information for a FieldAccess includes the object in which the field is stored, the field's name and type, and so on. Context information is captured by code defined in classes separate from the instrumentation; we have FieldAccessContext, BasicBlockContext and so on. This separation is significant; it leads a *pull-style* access to context information, rather than the *push-style* of other frameworks. Pull-style access brings modularity advantages; we return to this issue shortly.

New events by composition Many events are defined compositionally. For example, the library defines an Allocation event in terms of the various Java bytecodes that perform allocation: not only the new bytecode, but also newarray, anewarray and multianewarray.[18] Therefore, Allocation is defined as the union of several events. Users may define new events in a similar compositional fashion. (They may also define new events from scratch, at the bytecode level; we describe this in §3.6.)

Custom events and custom context User-defined events may require user-defined context information. For example, just as BasicBlockContext defines methods for getting the size and identifier of the basic block, and AllocationContext defines a getAllocated() method returning the allocated object, other events may wish to extract other information. These two examples represent different kinds of context: BasicBlockContext is said to be *static context*, because it can be determined at load time, simply by inspecting the bytecode and constant pool of the instrumented code. Meanwhile, AllocationContext requires inspection of the program state (to get the allocated object from the operand stack), so is *dynamic context*.

Push versus pull Custom context is a unique feature of our system: we are aware of no other system which allows users to extract additional information at an instrumentation site without also redefining the instrumentation site itself. For example, in Chord or RoadRunner, we would have to write new code against the underlying ASM or Javassist interfaces to achieve this; in doing so, we would

[18] In fact, we use ObjectConstructor, the event of executing the method body of java.lang.Object, rather than the new bytecode, because this also catches some allocations within the JVM or native code.

```
1  @Before(marker = BytecodeMarker.class, args = "getfield,putfield,getstatic,putstatic")
2  public static void onFieldAccess(BytecodeStaticContext bcsc, ClassContext cc,
3      FieldAccessStaticContext fasc, DynamicContext dc) {
4    Object owner; boolean isStatic;
5    int access = bcsc.getBytecodeNumber();
6    if(access == Opcodes.GETFIELD) {
7      owner = dc.getStackValue(0, Object.class);
8      isStatic = false;
9    }
10   else if(access == Opcodes.PUTFIELD) {
11     owner = dc.getStackValue(1, Object.class);
12     isStatic = false;
13   }
14   else {
15     owner = cc.asClass(fasc.getDeclaredOwner());
16     isStatic = true;
17   }
18   /* method continues ... */
19 }
```

Fig. 5. Non-reusable DiSL code for extracting the containing object for different cases of field access

extend the callback signature or Event data type used to pass the information to the client. This is push-style access to context information; the callback or Event definition is widened to encapsulate (hopefully) all information the client might require (else pay the cost of invasive code changes). Pull-style access, by contrast, avoids this need to fully anticipate what context information will be required. Rather, an unmodified instrumentation can effectively be extended simply by defining new context providers. For example, we could extend AllocationContext to provide a getSize() method returning the size of the object allocated, without changing the instrumentation defined by Allocation.

Abstraction over instrumentations A second benefit of our separate context approach is that we can abstract over multiple instrumentation sites, extracting commonality in terms of their context information. Our Allocation example is already an instance of this: many different bytecodes do allocations, and the method for getting a reference to the allocated object varies a little. In previous systems such as DiSL, with support for user-defined events ("markers") but not custom dynamic context, hand-rolled code would be required. Fig. 5 shows a typical example of this, from an implementation of the Racer race-detection algorithm [4] in the original DiSL. Since this hand-rolled code is written inside the snippet, it is not easily re-used. By contrast, in FRANC equivalent definitions can be in library code (namely FieldAccessContext).[19]

3.4 Separating Mappers from Updaters

As described earlier, a "thin waist" hourglass design is used to separate mappers (like FieldMapper) from updaters (like PostIncrement). We illustrate the key

[19] Although we could put the snippet shown in Fig. 5 into a library of such snippets, such a snippet could only be made useful by adding the "push" limitation. To pass on the extracted context information, here owner, would require adding code to make a method call, bringing the same fragility already noted in Chord and RoadRunner.

```
// shadow state
Map<Pair<CCTNode, String>, AtomicLong> state = new ShadowMap<>(...);

// subordinate mapper which maintains call chains in a calling context tree
final MethodCCT methodCCT = new MethodCCT(); // from the library

// primary mapper: latch the field being accessed, pairing it with current CC
class FieldCCMapper
  extends ThreadLocal<Pair<CCTNode, String>> // 1. maintain per thread <CC,String> pairs
  implements AfterCompletion<FieldAccess> {  // 2. by hooking field accesses
    public void afterCompletion(FieldAccess codeRegion) {
      set(Pair.valueOf(
        methodCCT.current.get(),             // 3. and sampling the call chain
        FieldAccessContext.getFullFieldName(codeRegion)
      ));
} }
FieldCCMapper currentFieldAndCC = new FieldCCMapper();

// update rule: atomic counter increment, from the library
Analysis<FieldAccess> updater = new PostIncrement<>(state, currentFieldAndCC);

// deployment -- infers any additional instrumentation needed
FRANC.deploy(FRANC.complete(methodCCT, currentFieldAndCC, updater));
```

Fig. 6. Composing a call chain sensitive field access counter

features of this design using a slightly more developed example. We stay with counting field accesses, as in Fig. 2, but wish to make it *context-sensitive*, in the sense of keeping counts per call chain (as well as per field identity). Fig. 6 shows such an analysis.

We note that our currentFieldAndCC (together with its equivalent in Fig. 2, currentField) is effectively a double indirection: it is a reference to an object containing a mutable reference. This is precisely what the standard Java ThreadLocal class implements, although in this case holding one mutable reference per thread. We provide our own ThreadLocal which rebases the standard Java implementation onto a new class hierarchy, in which it is a subclass of MutableReference. Our "thin waist" design is based on this class: the updater requires only that it has access to a MutableReference which records the current key object which it should use to index the map (which, again, can be any map).[20] The updater performs the lookup and updates the shadow value it finds.[21]

The map storing the results can equally well have either identity or equality semantics. However, we find identity to be a very useful option when mapping through complex domains, such as call chains in this example (or actually call chains paired with strings). Our library provides multiple tree-based data structures such as our MethodCCT (actually a prefix tree, or trie). These have the property that each node represents a distinct value (call chain), and the identity of the node suffices to signify that value (since each distinct call chain is represented by exactly one node). These prefix-structured key spaces are common in dynamic

[20] Here we pay a small price of not statically checking that the MutableReference yields a key object that is type-correct with respect to the map.

[21] We may compare this with the logical view presented in §2, where mappers were depicted as "routing" events from instrumentation to updater. Unlike routers, our mappers do not explicitly forward data; they simply maintain the state used by updaters to select which shadows to update. In effect, mappers implement the "control plane" of a router; the data plane resides in the updater and event subscription logic.

analyses of structured programs—consider not only call chains, but lock stacks, loop nests and others. Using these representations, key matching becomes a simple reference equality test, making identity-based maps the appropriate choice. This approach to representing values is similar to the interned string pool in the JVM or the autoboxing cache maintained by Integer.valueOf(n), and usually represents a more favourable time–space trade-off than content-based value comparisons. Furthermore, static context information (§3.3) is always stored in the instrumented bytecode's constant pool; strings stored here are always interned by the JVM, making such strings also suitable for use as keys in this way.

3.5 Supporting Common Case Usages: Autocomplete

Each instrumentation instance has a number of *subscribers*: the mappers and updaters that run in response to the instrumentation-generated events. FRANC.complete() infers, given a set of subscribers, any extra instrumentations that must be instantiated, and what subscription relationships to create.

As an example, in Fig. 2, we saw that no FieldAccess instrumentation was explicitly created, but two consumers of this information (both the mapper and the updater) were created. By passing these consumers to FRANC.complete(), the requirement for a FieldAccess instrumentation is inferred automatically.

FRANC.complete() is a variadic function. By deciding the order of the arguments, the programmer controls the ordering in which different subscribers' code will be executed. This is important because mappers and updaters interact by side-effecting updates of mapper state, and may both subscribe to the same event. Typically mappers should run first, so that an updater will make an "up-to-date" selection of shadow value. FRANC.deploy() warns if an updater is scheduled before a mapper. (We propose a more general approach to this issue in §7.)

3.6 Supporting Advanced Usages

We saw in §3.3 how custom context information can be extracted from code regions already defined. In some cases, we might wish to define not only new context information, but an entirely new code region. This is also supported. With this ability, we retain the full expressiveness of the underlying DiSL system. However, this functionality is applicable only in the rare cases in which existing code regions, or compositions thereof, do not satisfy the user's requirements. (One example might be superblocks or similar trace-like bytecode sequences, which are not currently implemented in our library.) To do so, the programmer must identify the bytecodes of interest using the underlying instrumentation API.

4 Case Studies

In this section we illustrate the benefits of FRANC by recasting two dynamic program analysis tools from the literature, Racer [4] and Senseo [27]. Racer

```
1  // shadow state
2  final Map<Pair<Object, String>, RacerState> state = new ShadowMap<>(...);
3
4  // mapper: latch the reference holding the field being accessed,
5  //         pairing it with the field name
6  class FieldOwnerMapper
7    extends ThreadLocal<Pair<Object, String>>       // 1. maintain a pair per thread
8    implements AfterCompletion<FieldAccess> {       // 2. by hooking field accesses
9      public void afterCompletion(FieldAccess codeRegion) {
10       set(Pair.valueOf(
11         FieldAccessContext.getHolder(codeRegion), // 3. and the field holder
12         FieldAccessContext.getFullFieldName(codeRegion)
13       ));
14 } };
15 final FieldOwnerMapper mapper = new FieldOwnerMapper();
16
17 // lock set analysis, from the library
18 LockSetAnalysis lockSet = new LockSetAnalysis();
19
20 class RacerAnalysis implements AfterCompletion<FieldAccess> {
21   public void afterCompletion(FieldAccess codeRegion) {
22     state.get(mapper.get()).onFieldAccess(
23       Context.getFullMethodName(codeRegion),      // location
24       FieldAccessContext.isFieldRead(codeRegion), // access type
25       lockSet.get()                               // set of held locks
26     );
27 } };
28 RacerAnalysis racer = new RacerAnalysis();
29
30 FRANC.deploy(FRANC.complete(lockSet, mapper, racer));
```

Fig. 7. FRANC recast of the Racer data-race detection tool

is a data-race detection tool, while Senseo is a dynamic analysis tool for code comprehension and profiling. Note that [2] describe different implementations of Racer based on AspectJ, while [23] compare implementations of Senseo based on DiSL, AspectJ, and ASM.

4.1 Racer

Racer is a data race detection tool for multi-threaded Java programs. Based on an extension of the Eraser algorithm [28], it reports a data race if two or more threads access the same field without holding any common lock, and if at least one of the threads is writing to the field.

The implementation of Racer addresses three major concerns. First, it maintains a mapping layer from field identifiers to their state. Note that the field identifier is a Pair<Object, String>: the first element identifies the object on which the field has been accessed, while the second contains the field's fully-qualified name. Second, it updates a thread local set of the locks held by each thread. Third, it intercepts field access events and updates some state corresponding to the accessed field. State information includes the threads that accessed the field, the intersection of the sets of locks held upon each access, and the access type (i.e., read or write). Fig. 7 illustrates the FRANC recast of Racer.

Lines 2–15 of Fig. 7 are dedicated to the first concern that we identified in Racer, that is, maintaining the mapping layer. In particular, line 2 defines a Map (i.e., state), while lines 6–15 define its updating function (i.e., mapper).

This mapping layer resembles those illustrated in Fig. 2 and Fig. 6. While the mapping layer in Fig. 2 statically identifies fields with a unique field identifier, the implementation presented in Fig. 7 takes into account also the reference on which

```
1 // mapper which maintains call chains in a calling context tree
2 final MethodCCT methodCCT = new MethodCCT(); // from the library
3
4 // (a) first define a stateful mapper which latches the current CC
5 class CCMapper
6   extends MethodLocal<CCTNode> {      // 1. maintains the current CC in a local variable
7     public CCTNode initialValue() {   // 2. setting its initial value
8       return methodCCT.current.get(); // 3. by sampling the call chain
9 } };
10 // (b) then instantiate it
11 CCMapper currentCC = new CCMapper();
```

Fig. 8. Mapping layer of the FRANC recast of Senseo

the field has been accessed. Note that it is possible to replace the mapping layer in Fig. 2 with the one in Fig. 7 without affecting the rest of the code. Moreover, it is possible to combine the mapping layer in Fig. 6 with the one in Fig. 7 by using Tuple<CCTNode, Object, String> as key.

Line 18 of Fig. 7 instantiates a lock set analysis component that is part of the FRANC library. This maintains a thread local set of locks held by each thread, thus addressing the second concern of Racer.

Finally, the third concern of Racer, that of updating the state associated to the accessed field, is addressed in lines 20–28 of Fig. 7. The logic for updating the analysis state is here an essential detail of the analysis, so is of limited reusability. However, we note that good modularity properties remain. For example, replacing the mapping layer would not require any modification to this part of the code.

4.2 Senseo

Senseo is a dynamic analysis tool for code comprehension and profiling. It collects context-sensitive dynamic information for each invoked method. This information consists of (1) the number of method executions, (2) the run-time type of method arguments, (3) the run-time type of return values, and (4) the number of allocated objects. Each of these can be seen as collected by a smaller sub-analysis. The implementation of Senseo addresses two key concerns. Firstly, it must maintain the current call chain for each thread. Secondly, it must perform the necessary analyses upon method entry, method exit, and memory allocation.

Fig. 8 illustrates the FRANC code addressing the first concern, that is, maintaining per-thread call chains. This code is equivalent to part of the code illustrated in Fig. 6. However, here we do not declare a single Map to store the shadow state. Rather, each specific analysis allocates its own Map. In this way, single analyses can be added or removed without affecting the rest of the code.

Fig. 9 illustrates the different analyses of which Senseo is composed. Each analysis is completely independent from the others, and could be easily disabled or extended. An advantage of this modular design is that adding an additional analysis, for example to count the number of basic blocks of code executed within each call chain, is as trivial as adding the code snippet presented in Fig. 10. Another advantage is that this code does not depend on a specific instrumentation of BasicBlock code regions, but it can be reused for any custom instrumentation that intercepts code regions that implement the BasicBlock

```
// shadow state for methodCalls
Map<CCTNode, AtomicLong> callsState = new ShadowMap<>(...);
// update rule: atomic counter increment, from the library
Analysis<Body> methodCalls = new BeforeIncrement<>(callsState, currentCC);
```

(a) Count the number of method executions

```
// shadow state for methodArgs
Map<CCTNode, ArgsProfile> argsState = new ShadowMap<>(...);
// runtime argument type analysis, from the library
Analysis<Body> methodArgs = new ArgumentAnalysis(argsState, currentCC);
```

(b) Profile the runtime type of method arguments

```
// shadow state for methodRets
Map<CCTNode, RetsProfile> methodRetsState = new ShadowMap<>(...);
// runtime return type analysis, from the library
Analysis<Body> methodRets = new ReturnValueAnalysis(methodRetsState, currentCC);
```

(c) Profile the runtime type of return values

```
// shadow state for allocs
Map<CCTNode, AllocsProfile> allocsState = new ShadowMap<>(...);
// allocation analysis, from the library
Analysis<Allocation> allocs = new AllocationAnalysis(allocsState, currentCC);
```

(d) Profile object and array allocations

Fig. 9. Different analyses in the FRANC recast of Senseo

```
// shadow state for bbs
Map<CCTNode, BasicBlockProfile> bbsState = new ShadowMap<>(...);
// allocation analysis, from the library
Analysis<BasicBlock> bbs = new BasicBlockAnalysis(bbsState, currentCC);
```

Fig. 10. Optional basic block analysis for the FRANC recast of Senseo

interface. This is particularly important in this case, as developers may want to use custom algorithms to define BasicBlock regions. For example, bytecodes that could throw an exception may (or may not) define the beginning of a new basic block.

5 Performance Evaluation

In this section, we evaluate the performance of the Senseo and Racer case studies reimplemented using the FRANC framework, and compare it to the performance of the same tools implemented using DiSL. We compare the execution times of both tool implementations on the benchmarks from the DaCapo [3] suite (release 9.12), excluding the tradesoap, tradebeans, and tomcat benchmarks due to well known issues[22] unrelated to our framework. We have used the default workload size to evaluate the Senseo tools, and the small workload size to evaluate Racer tools, due to high memory consumption with both the DiSL and the FRANC

[22] See bug ID 2955469 (hardcoded timeout in tradesoap and tradebeans) and bug ID 2934521 (StackOverflowError in tomcat) in the DaCapo bug tracker at http://sourceforge.net/tracker/?group_id=172498&atid=861957.

frameworks. All experiments were run on a multicore platform[23] with all non-essential system services disabled.

We present the results in Fig. 11, both for the startup and the steady-state performance. Each column in the plot corresponds to a single DaCapo benchmark, and we report the overhead factor of a FRANC-based tool over a DiSL-based tool, when applied to that benchmark. In the last column, we report the geometric mean of all overhead factors. The hollow data points correspond to the mean startup overhead, while the filled data points correspond to the mean steady-state overhead. The whiskers represent a 95% confidence interval for the means.

To determine the startup overhead, we executed 3 runs of a single iteration of each benchmark and measured the time from the start of the process till the end of the iteration to capture the instrumentation overhead. We relied on the filesystem cache to mitigate the influence of I/O operations during startup. To determine the steady-state overhead, we made a single run with 15 iterations of each benchmark. Based on visual inspection of the data, we excluded the first 3 iterations to minimize the influence of startup transients and interpreted code.

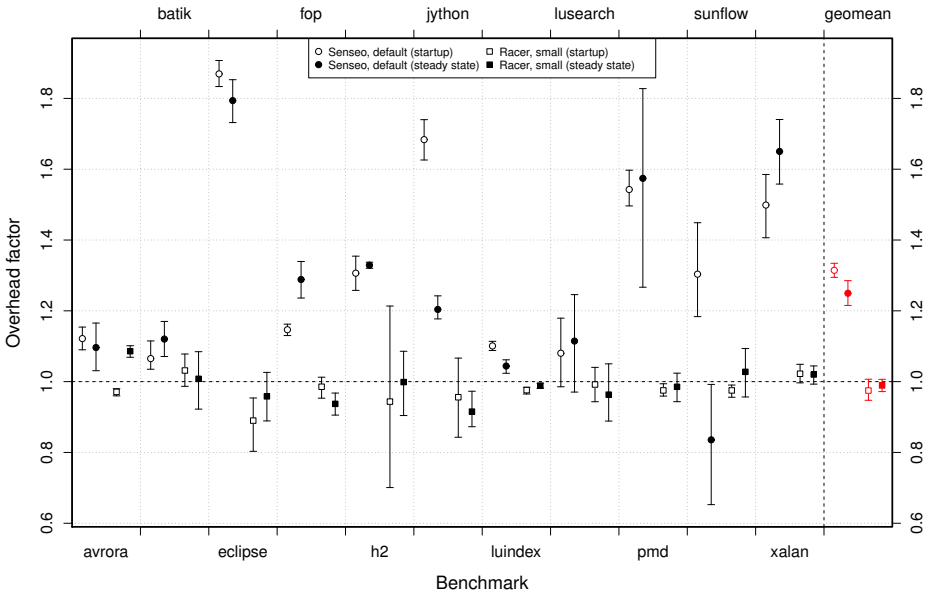


Fig. 11. Mean startup and steady-state overhead, with 95% confidence interval.

Since our system considerably raises the level of abstraction for the analysis developers, we expect to pay certain price in terms of performance. The additional

[23] Intel Core2 Quad Q9650 CPU, 3.0 GHz, 8 GB RAM, 64-bit Ubuntu GNU/Linux 12.04, kernel 3.2.0-20, Oracle Java HotSpot 64-bit Server VM 1.6.0_32, 7 GB heap.

complexity of the inlining of event subscribers may cause additional overhead at instrumentation. The composability of elements and the separation between shadow value mapping and updating inevitably leads to more indirections at run time, increasing overhead.

The results confirm our expectations. While on average, the FRANC-based Racer tool performs on par with the DiSL-based implementation, the FRANC-based Senseo tool shows approximately 31% startup, and 25% steady-state overhead. The average is not entirely representative in the case of Senseo, because there are 3 benchmarks (eclipse, pmd, and xalan) showing considerably higher steady-state overhead, and 4 benchmarks (also including jython) showing considerably higher startup overhead. The absolute worst-case overhead was observed in the eclipse benchmark under the FRANC-based Senseo tool, which yielded an 85% upper bound for the confidence interval.

However, we note that the system is still in a prototype stage, with opportunities for optimization (actually enabled by the higher level of abstraction) still unexploited. A key benefit of our design is that it has been carefully crafted to compile down to code very similar to what an analysis developer would write by hand using a more traditional instrumentation system, such as DiSL (which serves as the foundation for our system). Considering that the DiSL-based Senseo tool was more than twice as fast when compared to an AspectJ-based implementation [23], we consider the overhead found in this evaluation acceptable, and expect the cost of the higher abstraction level to remain in reasonable bounds.

6 Related Work

Basic instrumentations Several instrumentation systems including Pin [21] and the instrumentation engine of Chord provide callback-based APIs which allow programmatic construction of analyses in response to a closed set of events. As described in §2, these systems are valuable, but inadequate to modularise complex analyses. DiSL [23] provides slightly more flexibility in defining new instrumentations and extracting additional data ("context information"—§3.3) but retains the same key limitations. Aspect-oriented languages such as AspectJ [18] or AspectC [8] provide analogous support, in the form of join points, but these are again fixed by the system, and lack the low-level (instruction or basic block) join points provided by instrumentation systems.

Event and shadowing abstractions in RoadRunner RoadRunner is a dynamic analysis framework highly specialised towards data race detection and related analyses. Only per-field, per-lock and per-thread shadowing is provided. Adding new event types would require changes to core interfaces, including the basic Tool superclass. By contrast, our system supports an open set of events and a generalised approach to shadowing based on shadow mappers (§3.2), and provides for many common requirements using library code, making it more general but no harder to use. RoadRunner decomposes analyses into an event-processing pipeline, admitting some modularity and re-use. The linear (pipeline) topology

of data flow in RoadRunner is also limiting, because decisions made by one pipeline stage affect all subsequent stages; there is no provision for "forking" of pipelines or "fan out" of events, yet many of the more complex use cases require this. (Consider an analysis for comparing the performance of different locking disciplines; we would want to synthesise two distinct streams of locking events, each in a manner similar to RoadRunner's existing FaultInjection module, then simulate the performance of each. Unfortunately, RoadRunner's interfaces provide no way to distinguish the two separate streams of lock events.)

Relations in Chord The Chord framework provides two support mechanisms for the creation of dynamic analyses. The most basic is a set of instrumentation-based callbacks, implemented on top of Javassist [6, 7], as already discussed. The second is a relational storage and query model (based on Datalog) which can be used to store and analyse collected data. These additional abstractions assist the programmer in a manner roughly analogous to our mapper and updater facilities. Each different kind of incoming event can be seen as a distinct "program domain" (in Chord terminology), shadow value maps as Datalog relations, updaters as updates to the contents of a relation, and mappers as join-based queries selecting the elements to update. However, an important difference is that Chord's storage and query infrastructure is shared with its static analysis capabilities, and consequently incorporates far-reaching design choices optimised for scalability of static analysis problems to large input programs. In particular, all analysis happens in a postprocessing stage, rather than being interleaved (or in parallel) with the base program, since relations cannot be queried until each participating domain is fully constructed. For example, a simple profiler counting basic block executions by call chain would defer computing counts until the domain of call chains was fully constructed, i.e. until the end of execution. Although justifiable within static analysis problems, when used for dynamic analysis they exclude a key mode of application: interactively monitoring running systems.

DTrace DTrace [5] is a system for dynamic, safe observation of both user and kernel code in production systems. It permits an open set of events (or "probes" in DTrace terminology), defined separately from the framework by distinct *providers* having user-defined semantics and implementations. Implementing a new provider is a very involved process; indeed, one plausible approach could to use a dynamic instrumentation system of the kind we have discussed. DTrace's built-in user-level provider eschews existing dynamic instrumentation systems in favour of source-level macros and link-time interposition, largely to ensure a *zero overhead when disabled* property. (We note that this is a property of a provider, not of DTrace in general.) Analyses in DTrace are written in D, an Awk-like scripting language including some powerful features for storing and aggregating collected data. These include built-in associative mappings, which may be keyed on various kinds of values, including call chains, and a notion of aggregation function which can be used to schedule update operations in a thread-local, contention-minimising fashion. However, D's containers are built-in and its data model is ad-hoc. Our approach is more flexible and more general: our associative containers can be

keyed not only on some fixed set of data types, but on any domain that can be constructed by the programmer (such as call graph edges, lock stacks, loop nests, objects, etc.). On the other hand, our system does not share the safety or performance properties that are key constraints on D's design.

Other domain-specific languages MDL [15] is a domain-specific language for describing performance metrics in terms of instrumentation snippets that compute them. It provides a closed set of join points (including procedure entry/exit and other common cases), *constraints* (turning on or off instrumentation based on the properties of a candidate instrumentation site), and *program resource* definitions which can restrict instrumentation to particular parts of the program. A limited form of composition is supported by combining new constraints with pre-existing, less specialised metrics—but only if that constraint was anticipated by the instrumentation author as being applicable. As well as offering limited compositionality, the system is highly specialised towards performance measurement.

7 Conclusions and Future Work

We have presented FRANC, an API which lifts dynamic analysis construction from the level of instrumentation to an event-based publish–subscribe system with convenient re-usable abstractions for data collection and aggregation.

Although embedding in Java has many benefits, the implicit data flow of an imperative language is a hindrance in many circumstances where we would like to reason explicitly about data flow—for example, to infer the appropriate ordering between listeners responding to the same event. A fully declarative approach is the logical next avenue to explore. We note that the "signal" abstraction from functional reactive programming fits neatly with our design: it has been argued as an improvement over explicit listener-style publish–subscribe systems [9, 24, 22], and our use of "current" values during shadow mapping (§3.2) is logically the act of sampling a signal. We plan to explore these synergies in the near future.

Acknowledgments The research presented in this paper has been supported by the Swiss National Science Foundation (project CRSII2_136225), by the Scientific Exchange Programme NMS–CH (project 11.109), by the European Commission (Seventh Framework Programme projects 287746 and 257414), by the Sino-Swiss Science and Technology Cooperation (SSSTC) Exchange Grant (project EG34–092011) and Institutional Partnership (project IP04–092010), and by the Czech Science Foundation (project 201/09/H057).

Bibliography

1. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. ACM SIGPLAN Conf. on Programming language design and implementation*, pages 85–96. ACM, 1997.

2. D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Parallel dynamic analysis on multicores with aspect-oriented programming. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 1–12. ACM Press, March 2010.
3. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. 21st ACM SIGPLAN Conf. on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06)*, 2006.
4. E. Bodden and K. Havelund. Racer: effective race detection using AspectJ. In *Proc. Int. Symp. on Software Testing and Analysis*, ISSTA '08, pages 155–166. ACM, 2008.
5. B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proc. USENIX Annual Technical Conference*, ATEC '04, page 2. USENIX Association, 2004.
6. S. Chiba. Load-time structural reflection in Java. In *Proc. 14th European Conference on Object-Oriented Programming*, pages 313–336. Springer, June 2000.
7. S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. *Lecture Notes in Computer Science*, 2830:364–376, 2003.
8. Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectC to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, 26(5):88–98, Sept. 2001.
9. A. Courtney. Frappé: Functional reactive programming in Java. In *Proce. 3rd Int. Symp. on Practical Aspects of Declarative Languages*, PADL '01, pages 29–44. Springer, 2001.
10. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6. USENIX Association, 2010.
11. C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proc. ACM SIGPLAN Conf. on Programming language design and implementation*, pages 121–133. ACM, 2009.
12. C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Proc. 9th Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 1–8. ACM, 2010.
13. S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proc. SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 120–126. ACM, 1982.
14. M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: how to cheat and not get caught. In *Proc. ACM SIGMETRICS Int. Conf. on Measurement and modeling of computer systems*, SIGMETRICS '02, pages 140–151. ACM, 2002.
15. J. Hollingsworth, O. Niam, B. Miller, Z. Xu, M. Goncalves, and L. Zheng. MDL: a language and compiler for dynamic program instrumentation. In *Proc. Conf. Parallel Architectures and Compilation Techniques*, pages 201–212. IEEE, 1997.
16. M. Jovic and M. Hauswirth. Listener latency profiling: Measuring the perceptible performance of interactive java applications. *Science of Computer Programming*, 76(11):1054 – 1072, 2011.
17. M. Jump and K. S. McKinley. Dynamic shape analysis via degree metrics. In *Proc. Int. Symp. on Memory management*, ISMM '09, pages 119–128. ACM, 2009.

18. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353. Springer, 2001.
19. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symp. on Code generation and optimization*, CGO '04, 2004.
20. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, second edition, 1999.
21. C.-K. Luk, R. Cohn, R. Muth, H. Ptil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proce. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 191–200. ACM, 2005.
22. I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, EPFL, 2012.
23. L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. In *Proc. 11th Int. Conf. on Aspect-Oriented Software Development*, pages 239–250, 2012.
24. L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proc. 24th ACM SIGPLAN Conf. on Object oriented programming: systems languages and applications*, OOPSLA '09, pages 1–20. ACM, 2009.
25. N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation*, pages 89–100. ACM, 2007.
26. N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: generating program traces with object death records. In *Proc. 9th Int. Conf. on Principles and Practice of Programming in Java*, PPPJ '11, pages 139–142. ACM, 2011.
27. D. Rothlisberger, M. Harry, W. Binder, P. Moret, D. Ansaloni, A. Villazon, and O. Nierstrasz. Exploiting dynamic information in ides improves speed and correctness of software maintenance tasks. *Software Engineering, IEEE Transactions on*, 38(3):579 –591, 2012.
28. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.
29. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for c. In *Proc. 10th European software engineering Conf. held jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of software engineering*, ESEC/FSE-13, pages 263–272. ACM, 2005.
30. Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition?: DeFuse: definition-use invariants for detecting concurrency and sequential bugs. In *Proc. ACM Int. Conf. on Object oriented programming systems languages and applications*, OOPSLA '10, pages 160–174. ACM, 2010.
31. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, Jan. 1986.
32. J. Weidendorfer. Sequential performance analysis with callgrind and kcachegrind. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing*, pages 93–113. Springer, 2008.
33. J. Zukowski. *The Definitive Guide to Java Swing.* Apress, 3rd edition, 2005.