The Code of Many Colors: Semi-automated Reasoning about Multi-Thread Policy for Java

Dean F. Sutherland

CMU-ISR-08-112
May 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This material is based upon work supported by the following grants: NASA: NCC2-1298 and NNA05C530A; Lockheed Martin: RRMHS1798; ARO: DAAD190210389; IBM Eclipse: IC-5010. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied of the sponsor or the U.S. government.

Keywords: Java, Program evolution, Static analysis, Race Conditions, Multi-threaded programming, Design Intent, Concurrency, Program assurance

For Libby, who has spent far too many years providing front-line support for me and for this work. She's in serious danger of being recognized as the world's finest spouse. Also for Ivan, who has spent too many years and too many dollars providing financial support for my graduate schooling.
I couldn't have done it without either of you.

Abstract

Concurrent programming has proven to be difficult. One cause of this difficulty is that the relevant thread usage policy seldom appears either in documents or code comments. A second difficulty is that thread usage policy—even when it is known—imposes widespread consequences on the code to be written. Finally, finding and removing concurrency faults in existing code is hard.

This thesis introduces thread coloring, a language of discourse useful for concise expression of and reasoning about intended thread usage policies in a wide variety of code. Thread coloring addresses a range of concurrency issues—assuring single-thread access, identifying possibly-shared data regions and localizing knowledge about roles for threads—that have not previously been comprehensively addressed. Using this language, programmers can model design intent about relationships among the roles of threads with respect to segments of executable code and also with respect to shared state. Programmers formally link the model with their code by expressing the model as annotations in that code.

This thesis describes a prototype analysis tool, integrated into an integrated development environment, and its use in case studies to demonstrate that thread coloring is a feasible and practicable approach to expressing and understanding thread usage policies, including complex ones. The tool analyzes consistency between the expressed model and the as-written code, and notifies programmers of discrepancies between them. The case studies use published code to demonstrate that developers can express useful models, identify concurrency faults and assure policy compliance. The thesis includes a demonstration of scaling to a medium-sized program of 140KSLOC and a demonstration of the potential to scale to much larger programs and support composition among analysis results for separately developed components. By limiting the problem scope to thread usage policy, the prototype implementation requires one hundred times fewer annotations than are needed for full functional correctness—6.3 annotations per KSLOC, potentially reduceable in future by another order of magnitude.

This thesis provides five primary contributions to software engineering. First, it provides a language that developers can use to express thread usage policies. Second, it provides a systematic way to improve code quality by assuring that as-written code complies with expressed thread usage policy. Third, it uses a new combination of preexisting techniques to reduce the effort required to express models to very low levels. Fourth, it demonstrates techniques that permit the analysis to operate on very large programs—millions of lines of code appear to be within reach. Finally, it demonstrates techniques that permit straightforward and reliable incremental recomputation of results after a program change.

Acknowledgments

Every member of the Fluid group provided essential support and encouragement throughout my research. Edwin Chan kept the software working and pulled together whatever infrastructure I needed. John Tang Boyland built the IR and much of the other infrastructure. Aaron Greenhouse showed me how to write analyses that work with the IR. John and Aaron wrote the effects analysis that provides my connection with data. Tim Halloran got us hooked into Eclipse, and provided the Drop–Sea TMS. Elissa Newman helped me understand how to visualize my analysis. Bill Scherlis pointed us all in the right direction, provided sage advice as needed, and kept the funding flowing. I certainly couldn't have done it without you.

Jonathan Aldrich spent endless hours helping me get my head around small-step semantics. He also provided a valuable sounding board about many different technical issues, especially during the design of the module system. Thanks for your patience and help.

Several outside readers spent way too much time reading drafts of dissertation chapters and telling me what I'd done wrong. I owe many thanks to Aaron Greenhouse, Daniel V. Klein, Joseph M. Newcomer, and Ivan E. Sutherland.

I especially want to thank Elizabeth Sutherland for her tireless efforts reading, critiquing and editing my dissertation. Without her efforts I would still have four different names for the color environment, a modules chapter on the brink of incomprehensibility, and a guided tour that loses readers at every turn.

The members of my thesis committee provided critical feedback about my research and my dissertation. They also provided exceedingly prompt turn-around on a dissertation that really should have been out weeks before it actually was. Thank you all, for your valuable input.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Vision

Computer programs are not self-documenting. Thus, throughout the history of programming, programmers have felt the need to express additional information regarding their intent for the programs through models, documentation, and other specifications. Often these externally expressed models and specifications are of sufficient scope of utility and value that they have influenced the design of later generations of programming languages. An obvious example is the introduction of string handling into Fortran in lieu of arrays of small integers with associated documentation regarding their interpretation as strings. A more recent historical example is the introduction of explicit strong typing into programming languages, replacing information notations such as Microsoft's "Hungarian notation." Our vision is to accelerate progress towards a similar impact on thread usage policy—an expression of the relationships between threads and the roles threads perform with respect to execution of code and access to shared state in a concurrent program. Our approach has three elements: introducing a formal language for documenting thread usage policy in the source code, inventing an analysis that can perform an automated consistency check between the policy and the as-written code, and developing techniques to reduce the effort of policy expression to a level that is plausibly acceptable to programmers. We evaluate and validate our approach through case studies on a diverse sample of fielded code.

1.2 Thread usage policy

A *thread usage policy* is a set of rules that, if followed, are intended to assure correctness with respect to concurrency. For example, modern GUIs are implemented using an "event thread" that is the only thread that may execute the bulk of the GUI's code, access internal GUI state, and dispatch events to user call-back methods. These GUI implementations enhance performance and avoid deadlock potential by assuming single-threaded access to their internal data structures, and thus avoiding the need for acquiring and releasing locks ("synchronization" in Java parlance). The GUI therefore requires that clients access its state only through its methods and only from the GUI event thread. GUI implementations also require that there may be at most one GUI event thread at a time. These requirements are part of a thread usage policy.

Policy-based concurrency rules appear in many contexts beyond just GUIs. For example, Enterprise Java Beans requires that Beans may not start or use any thread other than the one from which they are invoked—this is a thread usage policy. Another example arises when users of a shared resource choose a "single-writer/multiple-reader" approach, and implement this approach using multiple threads. Their thread usage policy might be: "There is at most one writer thread at a time; there may be multiple reader threads. When the writer is executing, no readers may be executing. A writer thread may both read and write the resource; reader threads must avoid writing the resource."

We consider separately the challenges of expressing thread usage policy—the rules that a program must follow—and of assisting developers in following those rules and, in particular, of assessing consistency of implementations and intended policies. The policies chosen by the GUI developers can sometimes confuse client developers, who may, for example, unintentionally invoke a GUI-related method from the wrong thread and cause intermittent data corruption inside the GUI. Some thread usage policies address this problem by mandating the use of locks; others, such as the GUI pattern described above, mandate single-threaded access to certain shared resources. Indeed, in our case studies it is clear that there are many cases where developers are fully aware of the policies but make errors in implementation and then struggle to find and repair those errors.

The focus of this thesis is on expressing and reasoning about ad-hoc thread usage policy in a practicable and scalable way. There are, of course, diverse approaches to regulating concurrency in languages such as Java. Our specific focus is on "non-lock concurrency"—the roles for threads with respect to execution of code and access to program state. Our approach is intended to complement and augment the vast body of related work. Greenhouse [23, 27] addresses lock-based concurrency. Chord [42], KISS [49], RacerX [12], WARLOCK [57] and others [33, 30] use static analysis techniques to check for race conditions in lock-based concurrent programs. Race Free Java and others [5, 6, 15, 48] modify a language's type system to prevent the possibility of race conditions. Extended Static Checking, ESC/Java and Daikon [39, 37, 17, 18, 44], among others, address general functional correctness of programs.

1.3 The problem: policy-based concurrency is difficult to implement correctly

Modern software depends on concurrency. In Java, for example, it is impossible to write even a simple applet without understanding the underlying thread usage policy. When libraries use concurrency, they impose a thread usage policy on their clients regarding how and when threads of control are used. When clients fail to comply with this policy, they risk state consistency errors and other hard-to-find bugs. Policy-based concurrency is an issue for ad-hoc multi-threading generally, not just in Java; it also appears, for example, in the Windows GUI interface in the Microsoft Foundation Classes (MFC) for C++ [1][43].

Unfortunately, it is difficult for programmers to avoid errors related to threads. This is true both for novices and for experienced programmers. Many frameworks and libraries establish policies with respect to thread roles and multi-threading.

In Sun BugParade bug 4143834 [35], for example, a novice `AWT` programmer complains that his attempts to cause various background threads to sleep led to freezes in parts of his GUI. The programmer invoked the `sleep` method while executing in the `AWT` event thread, thus "freezing" his GUI. This bug shows a novice mistake in `AWT` usage, brought on by his failure to understand that GUI callback methods execute on the `AWT` event thread, a key feature of the `AWT` thread usage policy.

A highly experienced development team reported a different bug:

> "We found the source of the flashing in our UI. We discovered that we were periodically blocking the AWT thread...oops! Our app is really quite large, and that made it rather difficult to isolate the cause of the problem" [47].

This expert team was well aware of the GUI's thread usage policy, but had violated it without realizing that they had done so. The blue flashes were a symptom caused by client code wrongly blocking the GUI event thread. Tracking down the fault took over a month.

These examples demonstrate that both novice and expert programmers have difficulty implementing code that uses policy-based concurrency. Three of the causes of this difficulty are

- Thread usage policy—even when it is known—imposes non-local consequences on the code to be written.
- Finding and removing concurrency faults in existing code is extremely challenging.
- The relevant thread usage policy is often unknown to programmers and may not be explicitly stated either in code or in documentation.

These problems are elaborated below.

[1]Policy-based concurrency also appears with multi-threaded programming and the Windows GUI generally. [43]

1.3.1 Respecting thread usage policy

Even when the thread usage policy is known, writing code that obeys that policy is difficult. Thread usage policies have implications that are widely distributed throughout the code. There are many questions that must be answered by the programmer: Does the policy permit me to touch that data from this method? Does it permit me to invoke that method from here? What threads might execute this portion of the code and should they be permitted to do so? These questions are specific instances of a more general question: How can I tell whether I am following the thread usage policy? The answers to these questions are often hard to find in documentation, and are often impossible to glean from inspection of code; programmers lack an effective feedback loop. To make matters worse, for many programmers, the need to even ask these questions may not be apparent. Yet the questions arise for *every* data reference and *every* method invocation that executes in the context of a thread usage policy.

Many multi-threaded frameworks assist programmers with thread usage policy by using a callback scheme. In such a scheme, a client might modify its data in some non-GUI thread, then notify the GUI that there is drawing to be done by calling the GUI's `repaint()` method, which may be called from any thread. Sometime later, the GUI invokes the client's `paint()` method from the GUI's event thread; this is a "callback." The `paint()` method manages drawing the updated data using GUI framework methods. This is permitted because the `paint()` method is executing on the GUI event thread. This callback example shows how unclear callback schemes can be about which actions occur on which threads. In this case, the GUI's `repaint()` service method may be invoked from any thread, but the client's `paint()` callback method may only be invoked from the GUI event thread. Many GUI frameworks fail to clearly document this confusing aspect of their thread usage policy; specifically, they fail to address which thread roles execute which methods.

In callback schemes of this kind, it is crucially important, but not obvious, that the client must avoid invoking its own `paint()` method directly. If the client does so, the `paint()` method would execute in the wrong thread, thus breaking the GUI's thread usage policy.

1.3.2 Repairing thread usage policy faults

Programmers have trouble debugging errors and failures in policy-based multi-threaded code, in part because the symptoms they observe may not be obviously threading–related. In particular, symptoms that may arise include

- Concurrent modification exceptions from Java iterators. This may arise in concurrent code when one thread modifies a `Collection` while another thread is iterating over the same `Collection`.
- Intermittent and seemingly random data corruption due to state consistency problems. This symptom could arise due to a race condition in concurrent code, such as when a GUI client invokes a GUI method from the wrong thread.
- Unexpected GUI pauses or freezes. Pauses could arise because client code mistakenly performs a lengthy computation on the GUI event thread; the GUI cannot service events while the computation is underway. A freeze could be a symptom of deadlock, perhaps caused when client code running on the GUI event thread attempts to acquire a lock that should be acquired only from non-GUI threads.
- Obscure problems such as occasional "blue flashes" in portions of the GUI. In [47] this was a symptom caused by client code wrongly blocking the GUI event thread. The developers originally thought that the problem was due to bugs in the GUI; tracking down the actual cause took over a month for a highly experienced team who were well aware of the GUI's thread usage policy. The total cost to diagnose and repair this bug was over one person-year.

Such problems can be notoriously difficult to debug. The manifestation of these errors and failures at runtime is often dependent on timing, context, load, and processor architecture. Change the conditions or the timing—by using the debugger, for example—and the symptom may change or disappear. Alternatively, failures or errors may occur on one out of each million executions of the faulty code; this is nearly impossible to isolate in a debugger, but still frequent enough to be an issue for long-running or frequently-run applications.

Because debugging these errors and failures is so difficult, programmers often fall back on code reviews to help locate the underlying faults in the code. However, code reviews have limited effectiveness when addressing non-local properties because the consequences of those properties are spread widely and thinly throughout the code. Furthermore, this non-locality also means that *every* instance of each consequence must be handled correctly; if even one is wrong, the program may fail.

1.3.3 Thread usage policy knowledge

To obey a thread usage policy, the programmer must know what that policy is. He may desire to "look it up," either in the code or in the documentation. But many APIs fail to clearly specify the intended thread usage policy to which clients must adhere. Even worse, a novice programmer may not even be aware that such a policy exists at all. For example, in [34] members of the `AWT` core development team write

> The AWT has never been completely thread-safe. Nevertheless, no official documentation has ever been written and released to developers stating this fact and encouraging developers to use a single-threaded GUI to avoid potential race conditions and deadlocks.

The AWT and Swing thread usage policy has been officially documented more recently, for example in [4]. Regardless, even when design documents are easily available, they may include misleading information because the thread usage policy has changed over time; see for example [41].

When the thread usage policy is not stated, a programmer may attempt to reverse engineer the policy from existing code, whether it be example code or code from his own organization. There are four problems with this approach. First, it assumes that the example code follows the thread usage policy correctly; this may not be a good assumption[2]. Secondly, thread usage policy decisions *are not directly evident* simply from reading example code. Although one can see what the code does, it is often impossible to distinguish accidental properties of the code from essential ones. That is, if one sees that some method is called only from the GUI thread, should one therefore conclude that the method may *not* be called from a *non*-GUI thread? Absent any expressed design intent, a programmer must make a guess, which may well turn out to be wrong. Thirdly, it may be necessary to examine and understand library source code in order to reverse engineer its thread usage policy. However, developers of client code often lack access to the source code for the libraries they depend upon. Finally, even when all of the prior difficulties are surmounted, deducing a thread usage policy from example code requires deep understanding of that code, which greatly increases the cost of this approach. All of this makes it extremely difficult to discover the required thread usage policy.

1.4 Our approach: Thread coloring

Our key idea is called thread coloring, which provides a language for expressing thread usage policy. Thread coloring specifications document the relationships between threads and the roles threads perform with respect to execution of code and access to shared state. We now describe each of the key players in these relationships.

Thread coloring specifications associate roles with threads. The distinction between thread roles and thread identity is exactly the same as the distinction between "The President of the United States" on the one hand, and "President George Washington" or "President Thomas Jefferson" on the other hand. The identity of the President changes every few years, but the office—the role—of President remains. Similarly, the identity of "the AWT event thread" may change during program execution. Nevertheless, there is at most one such thread at a time for any program.

Thread roles are the "colors" of thread coloring. Just as a person may act in many roles over his or her lifetime, so may a thread take on and lose many colors over its lifetime. A thread may act in more than one role (or color) simultaneously. Many threads can perform the same role simultaneously. The concept of thread colors allows us to discuss the maximum number of threads that may act in a given role at any one time—the *multiplicity* of the color. The interesting values for this multiplicity are zero, one, and "more-than-one."

Code segments are segments of executable code; the specific relationship expressed via the thread coloring language is "which thread roles, or combinations of thread roles, are permitted to execute a particular segment of code?" This property allows us to reason about which thread roles perform the actions represented by the code segment.

Data regions are identified portions of memory. They may include single objects, portions of single objects, portions of multiple objects, or other identified portions of a program's state. We use the definitions

[2]See Section 4.3 on page 76.

and infrastructure of Greenhouse–Boyland object-oriented effects analysis [24] to identify and describe data regions. The specific relationships expressed via thread coloring are "What thread roles, or combinations of thread roles, should be permitted to access a particular data region?" and "What thread roles, or combinations of thread roles, may access a particular data region?"

Programmers can use the thread coloring language to model the intended thread usage policy—*i.e.*, the relationship rules—in their code. They use annotations to express and formally link this thread usage model with the source code. Thread coloring analysis can then be used to check whether the model and the code are consistent.

The following sections provide brief descriptions of the thread coloring language and analysis. Chapters 2 and 5 present the language and analysis semantics in greater depth.

1.4.1 Formal Language

When design intent is informally documented as comments in source code, managing consistency of intent with the code can be done only by inspection. When design intent is formally expressed, consistency can be checked by an automated analysis. We have designed a thread coloring language using annotations placed in source code. These annotations allow a programmer to build a succinct model of intended thread usage; they express design intent that the programmer is—or should be—already thinking about. The annotations have a format that is similar both to the expressions seen in common programming languages and to the informal expressions programmers use to describe design intent at the white board. Notwithstanding their familiar expression, thread coloring annotations have precise meaning, and so can be checked automatically.

The thread coloring language supports

- Naming roles for threads. For example, the annotation "`@ColorDeclare AWT, Compute`" declares the existence of two thread roles named `AWT` and `Compute` for use in other thread coloring annotations.
- Identifying segments of code that require the executing thread to have a particular role, by placing an annotation such as "`@Color AWT`" at the beginning of the code segment. This annotation states the requirement that the current thread must have the AWT role at that program point for all possible program executions.
- Identifying segments of code that forbid the executing thread having a particular role, by placing an annotation such as "`@Color !AWT`" at the beginning of the code segment. The "!" indicates negation.
- Identifying data regions that require the executing thread to have a particular role in order to access the region by placing a "`@ColorConstrainedRegion Compute` **`for`** `MyDataRegion`". To forbid a role we use the same annotation as before, but this time with the role name negated as in "`@ColorConstrainedRegion !Compute` **`for`** `MyDataRegion`".
- Identifying execution points where threads acquire or lose particular roles, by placing "`@Grant Compute`" or "`@Revoke AWT`" annotations, respectively, at the relevant program points.
- Specifying the compatibility of various thread roles. The annotation "`@IncompatibleColors AWT, Compute`" declares that a thread may take on *at most one* of the named roles at any time; this is a global constraint.

Thread coloring annotations combine to build a model that provides a formal statement of design intent. This enables developers to check the design intent and the as-written code for mutual consistency.

The idea of lightweight and familiar-appearing annotations is not original to this work; the Fluid group[3] has demonstrated this general approach in prior work [27]. Similarly, our implementation of these annotations builds on infrastructure developed for that work. ESC/Java [16], JML [38] and Daikon [44] also use annotations in code to support analysis of program correctness. We suggest that our approach is "lighter-weight" in the sense that even with a small number of annotations, analysis can yield useful feedback to a developer regarding consistency of code and intent. However, our scope—thread roles—is more narrow than the broader scope of these tools, which focus generally on functional correctness.

[3]The Fluid group is the research project at Carnegie Mellon University of which the author has been a part.

1.4.2 Analysis

Thread coloring analysis operates in terms of *color bindings* and *color environments*. The roles performed by threads are represented by *colors*, which are names for those roles. The *color binding* of a thread at a particular instant of execution is the set of roles the thread is currently performing; this color binding may change during the course of execution. The *color environment* at a particular program point is a set representing all possible color bindings for all threads that may ever execute at that program point. Thus, a color environment is a set of sets of colors. An implementation could represent the color environment using a Boolean expression whose terms are color names; a positive term (*e.g.*, `AWT`) indicates inclusion of the named color, a negative term (*e.g.*, `!AWT`) indicates exclusion of the named color.

Thread coloring analysis proceeds within a method using symbolic execution to compute the current *color environment* at each program point within the method. Each method has a *color constraint*, representing the color bindings required to execute the method; the constraint also provides the initial color environment for the method body. At method invocation points, the analysis checks whether the current color environment satisfies the color constraint of all potentially invoked methods; in terms of boolean expressions, the test is $(\text{currentEnvironment}) \Rightarrow (\text{colorConstraint})$, where the $\Rightarrow$ operator is logical implication. At data access points, the analysis checks whether the current color environment satisfies the color constraint on all potentially accessed data regions.

The color binding of a thread changes when it executes at a program point where there is a `@Grant` or `@Revoke` annotation; thus the color environment at that point must change similarly during analysis. If these annotations may appear at any program point within the method, the analysis must operate on the control-flow graph of the method. Alternatively, an implementation could restrict these annotations to appear only on block boundaries and only in matched pairs. This latter restriction would permit a syntax-directed analysis, at the cost of somewhat reduced expressive power.

If all methods and all data regions in a program have explicit color constraints, and the consistency check at all method invocations and data accesses is successful for every method in the program, the analysis concludes that the program's as-written code and expressed thread usage model are mutually consistent.

Thread coloring analysis is firmly rooted in a long tradition of work on static analysis—see any modern compiler textbook for details. The form of the analysis is not new; we have, however, put the analysis techniques to new uses.

1.5 Assessing thread coloring

To assess the feasibility and practicability of thread coloring, we need to consider how the annotations and analysis might be integrated into a software engineering process. Annotating every method by hand is quite tedious even for a simple program. Similarly, performing the required analysis by hand is infeasible at even modest scale. We therefore built an analysis plug-in for the Fluid group's prototype assurance tool[4] to perform thread coloring analysis and thus to facilitate experimentation and evaluation with respect to feasibility and other key practicability issues. We now briefly describe the tool and our criteria for evaluation. A deeper discussion of the tool is seen in Chapter 6. Deeper discussion of the assessment criteria is located in Chapters 3, 4, and 6.

1.5.1 Tool

The prototype assurance tool processes annotated Java code and reports findings of consistency—or lack thereof—at every point in the code at which a thread usage policy applies. The consistency check assures adherence to the thread usage policy as expressed by the annotations. Of course, the expressed thread usage model may or may not be correct. The as-written code may or may not be correct. The tool has no way of telling whether the model is correct nor whether the code is correct. What the prototype assurance tool *can* do is to check whether or not each piece of annotated code is consistent with the model, and then report its findings to the programmer. The tool produces findings, whether positive or negative, for every method

[4]The Fluid group's prototype assurance tool is integrated into Eclipse. Thread coloring thus benefits from and builds on the Fluid group's infrastructure and prior work [24, 27, 59].

invocation and every “interesting” data reference. Interesting data references are those that may refer to a data region that has expressed thread usage policy. Negative findings point at each trouble spot and give a specific explanation of the fault, thus assisting the programmer’s efforts to make his code consistent with the policy. This allows him to

- Assure that his API clients conform to the API’s intended thread usage policy.
- Assure that his future code modifications remain consistent with the API’s expressed thread usage policy.
- Assure that should the API change its expressed thread usage policy, the tool will identify where his code is no longer in compliance.
- Assure that library code conforms to its own API’s expressed thread usage policy, when library source code is available (*e.g.* for the developers of the library).

In the process of consistency checking, the tool also provides local knowledge about non-local properties such as a method’s color environment. This local knowledge proved to be of great assistance while attempting to reverse engineer other people’s code in some of the case studies (notably JHotDraw; see Section 4.6 on page 100).

1.5.2 Criteria for practicability

Investigating and improving practicability is a key issue because practicability represents a step towards real-world impact. The prototype assurance tool was created for the purpose of evaluating the concepts of thread coloring with respect to diverse practicability criteria. These include scalability, the expressiveness of the thread usage model, actionability of results, and other issues. Besides usability by working programmers, perhaps the most important of these criteria relate to scalability, which has at least two facets. The ability to compose separately developed modules contributes to scalability with respect to team size as well as overall size of the system. Scalability also includes the size of individually analyzed components, which determines the range of analysis granularity.

Composition

Composition is the best way not only to scale-up but also to divide effort among developers. Because thread coloring analysis treats annotations as cut-points, programmers can annotate APIs and evaluate compliance with library threading policies. Thread coloring analysis handles large programs by assisting programmers to divide their programs into modules—each module containing anywhere from a few classes to thousands of classes—and to place thread coloring annotations on those module’s interfaces with each other. The prototype assurance tool then analyzes each module individually, in a composable and scalable manner. The module system is not specific to thread coloring; it supports two other analyses at this time, and can support a diverse group of analyses in the future.

Scalability

The core thread coloring analysis is, or can be, fundamentally linear in program size, which would appear to indicate no problems with scaling to large programs. That core analysis assumes, however, that the user has provided thread coloring annotations for every method in the program—an unsustainable burden. We have reduced the effort required by using annotation inference. However, this technique brings with it non-linear cost. Our solution is to use modules as our unit of composable analysis. We then require the programmer to annotate the module’s APIs, thus allowing the analysis to treat them as analysis cut-points. The analysis performs annotation inference inside each module, thus reducing the programmer’s model expression effort. Dividing the program into separately analyzable modules reduces the impact of the potentially worse-than-linear inference cost when compared to whole-program analysis; it also provides a scalable approach to analysis of programs that would be far too large for whole-program analysis on any reasonable hardware platform.

Using these methods, we have demonstrated composable scalability to moderate-sized programs of 140KSLOC; we have also demonstrated that we can analyze that same 140KSLOC as a single chunk. There appears to be no reason why we can't scale to large programs (*e.g.*, millions of lines of code), working with one module of perhaps 100KSLOC at a time.

Model expressiveness

Our goal is to make the thread coloring language capable of expressing typical thread usage policies, including complicated ones. The language has sufficed for almost all thread usage policies encountered to date. This includes both simple policies such as the standard GUI patterns, along with more complex policies such as Reader/Writer threads. We have demonstrated scale both in terms of numbers of thread roles in a policy (see Section 4.4) and in terms of the size of the code covered by the policy (see Section 4.5).

The lone thread usage policy-related issue we have been unable to model to date comes from an application performing pipelined processing of a VLSI circuit (see Section 4.5). The basic threading policy in terms of methods and threads was easy to express, but the thread usage policy with regard to data regions was not. This application takes a portion of the circuit and processes it in the first pipeline stage. When processing is complete, that portion of the circuit is handed off to the second pipeline stage, which is implemented using a different thread. The desired policy is that each pipeline stage has unique access to the portion of the circuit on which it is operating; when a stage passes a circuit-fragment to the next pipeline stage it passes all access-rights to the circuit fragment along with the data, keeping no copy or references for itself. Our inability to model this policy is not due to any deficiency in the thread coloring language. Rather, the problem is that the effects analysis that provides thread coloring's connection with data is not powerful enough to represent this style of usage. Modeling this data usage policy would require an approach more like ownership types [9] or fractional permissions [7].

Actionability of Results

A tool embodying these techniques should ideally provide results in a form that is directly actionable by a software developer. These results could include, for example:

1. Thread roles for code segments, *i.e.*, listing the set of roles of all threads that might execute a given segment of code.
2. An indication at every method invocation of whether or not all of the potentially invoked methods may be legitimately called from the current color environment, along with an explanation of why this is so.
3. An indication at every data reference of whether or not all of the potentially accessed data regions may be legitimately touched from the current color environment, along with an explanation of why this is so.
4. A combined result for all available annotated code indicating whether or not it complies with all stated thread usage policies. This result is the combination of the results from numbers 2 and 3 above.
5. A computed accessing color environment for each data region, indicating the possible combinations of thread roles that may touch that data region. This is the per-data-region roll-up of the individual items in result number 3 above.
6. Programmers can use the computed information from number 5 in combination with their knowledge of the multiplicity of the various thread roles to determine whether a particular data region is confined to a single thread, or alternatively whether the region requires synchronization of some kind.
7. The thread coloring language permits programmers to express a color constraint for a data region. This constraint can express single-threaded access. The prototype assurance tool reports any violations of the constraint.

We evaluated the results produced by the prototype assurance tool using a number of case studies drawn from external sources. Some of these case studies took place in the field, where we assisted practicing developers with analysis of their fielded code. During these case studies, the third-party developers had specific questions about their code: Is this data confined to a single thread? Does my code conform to the thread usage policy of the framework it engages with? What is that thread usage policy? Does this data require synchronization? The prototype tool provided results as described above, with the exception of determining thread confinement for data. In this latter case, the tool provides the information a user needs to reason about thread confinement, but does not itself perform the analysis.[5]

Other practicability issues

To be practicable, the model expression effort for thread coloring must be low enough that practicing programmers might plausibly consider adopting it. More specifically, it must provide the developer with a return on investment (ROI) that meets his needs in terms of both cost-benefit analysis and timeliness of benefit. The cost-benefit analysis must result in a perceptible overall benefit to the developer. In addition, the developer must realize significant benefit before his next schedule deadline; otherwise the developer will put off the work, and the tool will never be adopted. Practicing programmers always face a deadline.

Achieving a favorable developer ROI requires low expression effort, incremental adoption with early gratification, end-user comprehension and ease of use. These solution attributes are necessary for practicability and so for adoption in the real world; they may not be sufficient.

Reducing model expression effort The cost of expressing thread coloring models should ideally be low enough that practicing programmers would plausibly be willing to adopt the technique. Because we have not performed formal user studies, we approximate this measure by counting annotations required for model expression and comparing this number to the overall size of the code. We also record anecdotal statements volunteered by practicing programmers during case study engagements.

To decrease the effort required for annotating and analyzing programs, we have developed a semi-automated tool that acts as the programmer's assistant. The programmer inserts his annotations in a few strategic places. The prototype assurance tool reduces the cost of model expression via several techniques: annotation inheritance from ancestor classes, the use of scoped annotations to spread single payload annotations across many program points simultaneously, and finally annotation inference within modules to infer what the rules must have been for the rest of the program. In combination, these techniques led to annotation counts several orders of magnitude lower than those seen in other systems[6] [16, 17, 44, 45]. The required number of annotations per SLOC is low (*e.g.*, 6.3 per KSLOC), and might be made to be very low (*e.g.*, 0.25 per KSLOC) in future.

Annotation inference works best on large swaths of code, so we perform that inference within modules. As mentioned above, modules also allow us to avoid whole-program inference, thus maintaining scalability.

Incremental adoption model The Fluid group's policy is to avoid "big-bang" adoption [26]. Rather, we support an incremental adoption model in which each increment of model expression effort yields an immediate increment of benefit. That benefit may take the form of finding a bug, or it may take the form of expanding an *island of assurance* in the source code.

Islands of assurance are portions of the source code for which a programmer has provided design intent and within which the tool has assured compliance with that intent. This assurance is *contingent* on the assumption that unannotated code obeys the constraints at the edges of the islands of assurance. Thus, when problems related to thread usage policy arise, the developer should suspect the unannotated code outside of his islands of assurance.

The incremental adoption model works as follows: the prototype assurance tool doesn't bother the programmer about problems in unannotated code except at the boundary between unannotated and analyzed code. When unannotated code attempts to invoke a constrained method or to touch a constrained data region, the assurance tool reports an error, thus both diagnosing the problem and suggesting a location for the next increment of effort.

Developer comprehension and ease of use Developers who cannot understand the basic ideas of thread coloring will not consider adopting the technique. Similarly, if thread coloring is too difficult to use, it is unlikely to achieve a favorable developer ROI. We avoided formal user studies. Rather, we collected

[5]A future upgrade to the tool could answer questions about thread confinement directly. This analysis was beyond the scope of this thesis.

[6]Note, however, that these other systems attempt to prove broader functional properties of the code, which is a more ambitious scope than that addressed by thread coloring.

anecdotal comments volunteered by developers while we assisted them with analysis of their code during case studies in the field.

Sophisticated and experienced developers were able to write their own thread coloring annotations after about fifteen minutes' training. Less sophisticated developers did not write their own annotations, but were easily able to describe verbally "what the annotation should say" using the terminology of thread coloring.

Sophisticated and experienced developers easily understood the prototype assurance tool's reported results; less sophisticated developers required assistance with understanding the meaning of the tool's messages. Every developer, including the author himself, struggled with finding the critical analysis results in the prototype tool's voluminous output. Omitting extraneous reports remains a fertile ground for improvement.

1.6 Value of thread coloring in development

There are many groups that would benefit from using thread coloring. Authors of framework client code struggle with understanding the framework's thread usage policy and with assuring compliance. Examples include authors of GUI clients in Java, C#, *etc.*, for popular platforms including Windows, Mac, *etc.*

Authors of libraries, frameworks and APIs would benefit from adding thread coloring annotations to the interface of their API, because those annotations would make their API more valuable to those developers of client code who use thread coloring. Furthermore, using the thread coloring language reduces the difficulty of communicating changes in thread usage policy to their client developers. When thread usage policy changes, the API developer can change the annotations that embody their design intent. Obviously, the use of formal language opens the door to the use of automated checks of compliance.

For distributed development teams, thread coloring can assist in thread policy compliance on both sides of the API, and thus across distributed groups, by documenting policies on one side and assuring compliance on the other.

Developers and organizations who use thread coloring will realize several benefits:

- Concise expression of thread usage policies will make those policies more apparent to client programmers.
- Static checking of consistency between expressed thread usage policy and as-written code will catch threading-related defects that current development practices miss. Further, the static checking will catch these defects earlier in development, which saves debugging effort and rework.
- Thread usage policy documentation will be kept up-to-date because the developers receive immediate benefit from having a current expression of that policy available.
- Formal expression of required thread usage models at API, library and framework interfaces provides valuable knowledge capture. Having the models available makes the frameworks and APIs more valuable to their clients.
- Developers can check compliance with thread usage policies without needing to run tests or debug seemingly-random data corruption and other difficult problems.

1.7 Thesis Statement

We restate here our approach to policy-based concurrent code, now following the form of a scientific argument.

This thesis introduces thread coloring, a language of discourse that is intended to support expression of and reasoning about intended thread usage policies for a wide variety of code in a practicable and scalable way.

We have four principal hypotheses:

1. The thread coloring language can provide a useful and concise expression of the thread usage policies identified above.
2. In particular, we hypothesize that thread coloring can help address a range of concurrency issues—such

as assuring single-thread access, identifying possibly-shared data regions[7] and localizing knowledge about roles for threads—that have not previously been addressed in a comprehensive and systematic fashion.

3. This thesis further hypothesizes that it is possible to build a tool based on thread coloring that realizes a useful and practicable technique for assuring consistency between the expressed model and the as-written code.
4. This thesis hypothesizes that with this tool, it is possible to reduce the effort required for annotation and analysis sufficiently both to use thread coloring on larger bodies of code, and to plausibly enable adoption by practicing programmers.

Additional hypotheses were added to address the criterion of composability. These are addressed in Section 1.7.2 on the following page.

1.7.1 Outline of the dissertation

The remaining chapters of this dissertation make these hypotheses more precise, present means by which they can be assessed, and summarize the results of those assessments.

1. We have developed a language, whose annotations concisely express thread usage policies. The annotations are described in Chapter 2.

   We have defined the static semantics of the thread coloring language. This definition, which is presented in Chapter 5, informed the design and implementation of the thread coloring analysis. Because the focus of the thesis is on practicability issues, we defer to later work meta-theoretic proofs of the analysis itself.
2. Thread coloring has broad utility. We have performed and evaluated case studies on a broad selection of code from a variety of application domains. These case studies came from three different windowing toolkits, various open source projects, and from fielded applications both commercial and scientific. Details of some of these studies are found in Chapter 4. The case studies presented include a toy example program (see Section 4.3) and a 140KSLOC VLSI design tool (see Section 4.5); applications built using three different GUI toolkits and an action-planning and optimization application that has no GUI at all (see Section 4.8); a graphical editor (see Section 4.6) and an astronomy application (see Section 4.4), among others.

   Thread coloring can help address a range of concurrency issues. The case studies show how thread coloring can help address the following concurrency issues and assurances:

   - Methods invoked in the “right thread,” more properly “correct thread role” (all case studies, but see especially Section 4.3)
   - Data regions touched only by the “correct thread roles” (see Section 4.7)
   - Detection of potentially-shared data regions (see Section 4.3)
   - What thread roles should “get here?” (all case studies, but see especially Section 4.3)
   - What thread roles actually may “get here?” (all case studies, but see especially Section 4.3)
   - What threads (or thread roles) are relevant? (all case studies, but see especially Section 4.3)
   - Localizing knowledge about roles for threads, by propagating the thread colors through the as-written source code. (all case studies, but see especially Section 4.3)
   - Assurance that data regions intended to have only single-thread access are in fact accessed from only one thread. (see Section 4.7)
3. We have built a prototype tool as an existence proof. Chapter 6 discusses its implementation and some of the lessons we have learned while building it and experimenting with it. Specifically, we undertook a number of case studies which are detailed in Chapter 4. The prototype assurance tool provides thread usage policy assurances that are not otherwise available. In the case study in Section 4.7, these

[7] A similar hypothesis might address synchronizing access to shared state. This has already been addressed in [27] and elsewhere.

assurances have proven that certain data regions are in fact confined to a single thread, thus allowing programmers to remove all synchronization for them. The tool has found inconsistent thread usage in widely used programs, both commercial and open source (see Chapter 4). Anecdotal evidence from developers suggests that they highly value these results.

4. We have experimented with work-flow issues, user-interface issues, and support for reducing annotation and analysis effort. We have demonstrated (see Section 3.5) that it is possible to add thread coloring annotations to a moderate-sized application of 140KSLOC using only 6.3 annotations per KSLOC, and have designed additional improvements that would reduce this to 0.25 annotations per KSLOC.

   We have performed a pair of scalability studies—on Sky View Café (a ~25KSLOC shareware Java application; see Section 4.4) and on Electric (an ~140KSLOC open–source VLSI design tool; see Section 3.5). These scalability experiments suggest that the user-experience provided by adding thread coloring annotations and using the prototype assurance tool is interestingly light-weight and that the algorithms used in the analysis have the potential to scale up to much larger applications. Algorithmic performance is discussed in Chapter 6.

1.7.2 Modules

We needed to create some additional program analysis infrastructure, in the form of a module system, to address the challenge of composability. We describe here the hypotheses associated with this requirement. The motivation for modules is that analysis developers need to divide large programs into smaller pieces, analyze those pieces, and then compose the results. It is also useful for the pieces to be of significant size to enable inference-based algorithms to operate inside the pieces. Modules serve this role. Software developers need modules to support composable analysis in their software systems, and as a mechanism for controlling visibility and interfaces between the parts of their systems.

It seems surprising that we found it necessary to create another module system at this late date. Our module system focuses primarily on visibility and supporting composable analyses; the Java package system, by contrast, focuses on name-space management, and only partially on visibility.

The module system we developed provides explicitly identified module interfaces, a default hierarchy of modules for easy visibility control, backwards compatibility for code that is not placed into modules, module-at-a-time analysis support, and sealed modules. Modules that are "sealed" are complete when presented for analysis; their contents will not change during execution. The module system is intended to meet the needs of both analysis developers and Java programmers. For a deeper discussion of the design of the module system, see Chapter 3.

Hypotheses for modules

Because we found ourselves required to develop a new module system, we offer here some hypotheses regarding that design that are assessed as part of this research. In order to show that thread coloring is composable and scalable, we must also hypothesize that

- It is possible to split a large program into modules, treating the module boundaries as analysis cut-points.
- It is possible to analyze modules individually and to recombine the separate analyses so as to effect analysis of the entire program.
- Performing annotation inference inside each module, using the module boundary annotations as a starting point, will reduce the effort required for thread coloring annotation sufficiently to make the cost of modularization worth-while.
- Annotation inference within modules will significantly reduce the overall number of annotations to be written.

These hypotheses are considered in greater detail in Chapter 3.

The module analysis is part of the Fluid group's prototype assurance tool, which is integrated into the Eclipse integrated development environment. The module analysis thus benefits from and builds on the Fluid group’s infrastructure and prior work [23, 26, 59].

Chapter 2

Informal presentation of the coloring model

2.1 Introduction

This chapter introduces the basic concepts of thread coloring to a pragmatic programmer. The deeper discussions in the remainder of this dissertation assume the reader has understood these basic concepts. By the end of this chapter readers will know why they might need thread coloring, what it can do for them, and roughly how much work they may need to do in order to get useful results.

The chapter begins with a discussion of the motivating problems that led me to develop thread coloring, followed by a brief discussion of how it addresses those problems. The bulk of the chapter takes the form of an informal guided tour of the thread coloring process, using the GraphLayout Applet as a running example. The chapter closes by discussing issues of scale in terms of program size, numbers of thread roles, and overall model complexity.

2.2 Why use thread coloring?

Modern software depends on concurrency. Many APIs require their clients to adhere to strict thread usage policies in order to guarantee correct operation. When clients fail to comply with these policies, they risk difficult-to-debug state consistency errors and other hard-to-find bugs. Simply reading the buggy client code is rarely sufficient to discover or understand what *kinds of threads* there are—that is, their *roles*—much less what operations each thread role is—or should be—doing.

Unfortunately, many APIs do not clearly specify the intended thread usage policy to which clients must adhere. Reading the source code of example applications often fails to clarify the thread usage policy. Although one can see what the code does, it is often impossible to distinguish accidental properties of the code from essential ones. That is, if one sees that some method is called only from the GUI thread, should one therefore conclude that the method may *not* be called from a *non*-GUI thread? Absent any expressed design intent, a programmer must make a guess, which may well turn out to be wrong. Furthermore, it may be necessary to examine and understand library code in order to reverse engineer the thread usage policy. But developers of client code often lack access to the source code for the libraries they depend on. All of this makes multi-threaded code quite difficult to program successfully.

For example, one way Java supports GUI and graphic development is via the AWT. The AWT itself is not multi-threaded, but it does execute in its own thread and prescribes rules for how non-AWT threads may interact with it. These rules are not obvious; even experienced developers run into problems with the AWT thread usage policy. Here is a brief description of the salient points of the policy (from [4]):

1. There is at most one “AWT thread” per application.
2. There may be any number of separate “Compute threads.”

3. A Compute thread must never attempt to paint or to handle AWT data structures or events. Failure to comply can lead to exceptions from inside the AWT, because the AWT avoids both races and lock usage by accessing its internal data structures from within a single thread.
4. The AWT must "never" compute. Brief computation is acceptable, but extended work is not. Computing in the AWT thread leads to freezing the GUI until the computation finishes.

These policy decisions *are not directly evident* from reading example Applets. Even worse, the Fluid group discovered that GraphLayout, a widely distributed example Applet includes a subtle concurrency bug (see Sections 2.3.14 and 4.3). Yet, even when design documents are easily available, they may include misleading information because the thread usage policy has changed over time; see for example [41].

The Fluid group[1] believes that there are two key reasons why design intent is often missing or out of date [27]: the relevant intent can be difficult to express and there is little *immediate* benefit to keeping it up to date. First of all, many programmers are more comfortable with writing concise semi-formal annotations than with writing extensive prose descriptions of design intent. Using a familiar form of expression for design intent may reduce the perceived difficulty of that expression. Secondly, the short-term incentives for developers and for the organizations they work in are focused around meeting the next deadline; longer term issues often go by the wayside. When code is new, the developer is likely to remember the design decisions he has made. Thus time spent on design documentation does not directly help him meet his closest deadline. Similarly, time spent updating design documents is unlikely to pay off in the short term. The lack of documented design intent may only become a problem later, when even the original developer no longer remembers the details of his own design decisions (see Section 4.8).

2.2.1 The Fluid solution

The Fluid group pursues analyses that can be adopted incrementally and that provide immediate benefit for each increment of design intent expressed by the programmer. The immediate benefit may take the form of finding a hard-to-find bug. Alternatively, it may take the form of expanding an *island of assurance* in the code. These islands of assurance are sections of code that have been successfully analyzed; the prototype assurance tool has thus provided a contingent proof of the absence of certain kinds of problems within each island.[2] In the case of thread coloring, the problem we address is ensuring that actions occur on the right "kind of thread"—that is, the thread must have the role required by the action. Should a threading problem arise in a partially assured program, the programmer knows that he should begin looking *outside* his island of assurance; the prototype assurance tool has assured him that the *inside* of the island is OK.

To address the difficulty of expressing design intent, the Fluid group emphasizes simple concise expression of the mechanical properties that we analyze. We place the design intent directly in the code; the design intent is easier to find because it is close to the code to which it relates. In the case of the thread coloring language[3], simple concise annotations are placed in the source code to express individual thread usage policies, identify which threads take on which roles, and identify data regions. For a given thread role, the thread usage model—expressed in the annotations—describes which data regions it may access and which code segments it may execute. For a given data region or code segment, the model describes which thread roles may simultaneously access the data or execute the code. The prototype assurance tool performs static consistency checking to ensure that the as-written code complies with the thread usage policy expressed in the model.

It is important to note that the prototype assurance tool checks for consistency between thread usage model and as-written code; it does not check for "Truth." In this sense, thread coloring is much like double-entry book-keeping—it catches inconsistencies, whether deliberate or accidental; it cannot detect a consistent error. Thus, if the prototype assurance tool reports an error, it is up to the programmer to decide whether the problem is in the model, in the code, or in both model and code.

After each successful compilation the prototype assurance tool checks the consistency of the as-written code with the expressed thread usage model. Thus, if either the expressed thread usage model or the code

[1]The Fluid group is the research project at Carnegie Mellon University of which the author has been a part.

[2]The proof is contingent on the assumption that the as-yet-unexamined code outside the island of assurance obeys the design intent that is expressed within the island.

[3]The thread coloring language consists of the annotations themselves, as well as both the rules about their placement in the source code and the meaning of each annotation when so placed.

changes, the developer is promptly notified. We hope this will encourage programmers to keep their thread usage models up to date.

Another benefit of the prototype assurance tool's consistency checking arises when the developers of frameworks and libraries annotate their APIs. Not only would this make thread usage policies explicit, thus presumably improving client developers' ability to write conforming clients, but also thread coloring reduces the difficulty of communicating changes in thread usage policy to the client developers. When thread usage policy changes, the API developer can change the annotations that embody the design intent. Obviously, formal annotations open the door to the use of automated checks of compliance. Specifically, by using a thread coloring tool, programmers whose client code fails to comply with the new policy will get explicit error messages at each trouble spot. Programmers whose client code does comply will receive assurance that their code obeys the new thread usage policy.

2.3 Guided tour of thread coloring

In this section I present a guided tour of thread coloring, similar to that which I would present to an engineer to get her up to speed at the start of a case study. Thus, much of the tour is written in a conversational form. I make use of a running example to illustrate the need for and usage of each feature of thread coloring, along with the annotations needed to make it work.

A typical scenario for an in-field case study would begin with a brief presentation covering the same territory as the preceding discussion. The remaining plan of attack would be to:

- Discuss the overall structure of the program to be analyzed with its developers. The outcome of this discussion is the identification of a few portions of the program for which analysis results would be of greatest interest. For the purposes of this guided tour, we will use the AWT and Swing framework and their interaction with a client Applet—GraphLayout, in this case.
- Tentatively identify a candidate module structure for the portions of the program we will analyze, and insert the annotations required to create those modules and their interfaces. For our guided tour, we will place all of GraphLayout in a single module.
- Declare the necessary thread colors.
- Annotate a few strategically placed methods and data regions to constrain which thread roles may execute those methods or which thread roles may access those regions.
- Check for consistency between the expressed thread usage model and the as-written code. Depending on the results of that consistency check, we add some more annotations, and consistency check again.

We've already completed the first step in this process. We'll visit each of the remaining steps in the subsections that follow. Each subsection presents a feature of thread coloring. The subsection then gives a brief description of what the feature does and where it may be placed in the source code. Finally, the subsection gives a brief rationale of why an end user might want to use the feature. In particular, each subsection answers the questions that an interested end user would probably ask:

- What design intent does this feature allow me to express about the relationships between threads, executable code and data regions?
- Why and when would I want to use this feature of thread coloring?
- What benefit do I get as a result of this increment of work?

During this tour, I use "thread coloring" to refer to the entire process of adding annotations to code in order to model thread usage policy. I use "code coloring" to refer to annotating code segments and "data coloring" to refer to annotating data regions.

2.3.1 Modules

In order to use thread coloring to analyze a program, or a portion of one, we need to place it in a module structure. For a more detailed discussion of modules and the full motivation for their use, see Section 3.4 on page 43. The prototype assurance tool places each compilation unit (CU) into some module—either a module specified by an `@Module` annotation, or the predefined `default` module if there is no annotation for the CU. Some of the relevant properties of modules are:

- Every module has a public interface, which is a subset of the Java entities in that module. Java entities in the public interface of a module are potentially visible outside that module; all other Java entities in the module are hidden from the outside.
- Visibility within a single module follows ordinary Java visibility rules.
- There is a `default` module, which holds all code not explicitly placed in some other module. The public interface of the `default` module consists of all Java entities in that module; this public interface is visible to all code in all other modules—as restricted by ordinary Java visibility, of course.
- The granularity of placement of Java code into modules is the CU.

We place a CU into a named module with the annotation `@Module module_name`. Java entities that are part of the public interface of a module are so marked with the `@Vis` annotation ("Vis" for "visible").

@Module <name> This annotation declares the existence of the named module, and maps the annotated compilation unit into the named module. The named module is the *home module* of that compilation unit.

@Vis <name>? This annotation marks the annotated Java entity as being part of the public interface of some module.[4] If the name is absent, it will be the Java entity's home module. As a convenience for end users, marking a class with an `@Vis` annotation has the additional effect of marking all `public` fields and members of that class as being `@Vis`; similarly, if any field or member of a class is visible, the class itself is visible as well.

@NoVis This annotation declares that the annotated Java entity is *not* part of the public interface of its home module – or in fact, of *any* module. Typically used to over-ride class level `@Vis` annotations for members that should not be visible, or to restrict visibility of a Java entity located in the `default` module so that it is visible only to other Java entities located in the `default` module.

```
/** @Module GraphLayoutApp
    @Vis */
public class GraphLayout ... {
  ...
  /** @NoVis */
  public void mouseDragged(MouseEvent e) {
    pick.x = e.getX();
    pick.y = e.getY();
    repaint();
    e.consume();
  }

  public void mouseMoved(MouseEvent e) {
  }
```

Figure 2.1: Code snippet from GraphLayout's class `GraphPanel.java`

Example To show how the thread coloring process works, we'll create a running example using the AWT and Swing frameworks and their interaction with GraphLayout, a standard Java example Applet. We add the

```
@Module GraphLayoutApp
```

annotation to every compilation unit in GraphLayout to place all of its code into a single module named `GraphLayoutApp`. These are the only module annotations required for our example. To illustrate the meaning and use of the `@Vis` and `@NoVis` annotations, let us further suppose that we desire to make all members and fields of GraphLayout part of the public interface of their home module (`GraphLayoutApp`),

[4]The "?" indicates that the bracketed item may be absent.

except that we do not wish to expose the `mouseDragged(MouseEvent e)` method. The `@Vis` annotation applied to the entire class makes all fields and member functions visible along with the class itself. However, the `@NoVis` annotation applied directly to the `mouseDragged(MouseEvent e)` method overrides this visibility, leaving us with exactly the result we were after. The `mouseMoved()` method, for example, is made visible. In Figure 2.1 we see a code snippet from GraphLayout's class `GraphPanel`, showing both the `@Module` annotation needed for our running example and also the added `@Vis` and `@NoVis` annotations we temporarily inserted to demonstrate their use; only the `@Module` annotation remains part of our running example.

Modules affect thread coloring in three ways. First, the public interface of a module serves as thread coloring analysis cut-points. Each method that is part of the public interface of a module must be annotated with a color constraint that in fact provides all needed information for analysis outside that module. These cut-points allow the thread coloring analysis to scale to very large programs by analyzing one module at a time. Secondly, the color inference algorithm runs in the context of a single module. The thread coloring annotations on the public interface of the module provide the starting point for the inference algorithm. Lastly, the public interface of a module identifies the majority of places where the end user may need to write thread coloring annotations.

2.3.2 Color names

Modeling thread usage policy requires the ability to name the thread roles discussed in the policy. In the GUI example mentioned in Section 2.2, we see the use of an informal name: the "AWT thread," sometimes also known as the "AWT event thread," or the "GUI event thread." We use the `@ColorDeclare` annotation to declare names for thread roles.

@ColorDeclare <color_name_list> This annotation declares the existence of the named thread roles. The `<color_name_list>` is a comma-separated list of simple names. The color names so declared have a fully qualified name based on the location of their declaration. The standard **`package`**`-info.java` file found in each Java package is an excellent location for declarations of color names.

Color names have no pre-determined semantics; they are identifiers whose semantics arise from their usage. Typical usage is to declare a thread color for each distinct role for a thread in the program being analyzed. Thus, for a program that uses threads for printing, background rendering, network I/O, and computing the itinerary for a traveling salesman, we would expect to see a declaration like `@ColorDeclare Print, BGRender, NetIO, Itinerary`. Note that any particular thread could perform more than one role simultaneously and thus might have more than one of these colors.

Example Returning to our AWT/Swing GUI example, we write:

```
@ColorDeclare AWT, Compute
```

to declare the existence of two thread roles of future interest to our model. This annotation declares the existence of an "AWT" thread role and a "Compute" thread role.[5] We intend to use the `AWT` color for code and data that "belong to" the GUI, and the `Compute` color for code and data that *do not* "belong to" the GUI. We place the declaration of the `AWT` and `Compute` colors in the **`package`**`-info.java` file for package `java.awt`; the colors' fully qualified names are thus `java.awt.AWT` and `java.awt.Compute`. As with Java code, you can use the fully qualified name anywhere; you can use the simple name any time the declaration is in scope.[6]

Color name declarations serve to document the thread roles that are both potentially present and of interest in the program containing the declarations. This documentation provides a modest immediate benefit to the end user by answering the question "What thread roles are there in this program?". The declarations also provide an indirect benefit—by naming the thread roles, we can continue on to express additional properties of the program's thread usage model.

[5] The declaration is silent about any further properties, some of which are described in later sections.
[6] Importing color declarations is described in Section 2.3.12 on page 30.

```
package java.awt;
class Button extends Component implements ... {
   /** @Color java.awt.AWT & !java.awt.Compute */
   public Button(String label) {...}
   ...
   /** @Color java.awt.AWT & !java.awt.Compute */
   public void setLabel(...) {...}
   ...
}
```

Figure 2.2: Code from `java.awt.Button`

2.3.3 Color constraints

A key use for color names is to constrain what thread roles may execute at a program point or what thread roles may access a data region.

Local color constraints

We use local color constraints to build the thread usage model by expressing rules such as "a `Compute` thread must never attempt to paint or to handle AWT data structures or events."

@Color <boolean expression> The `<boolean expression>` may in principle be any Boolean expression over color names; the prototype assurance tool supports only expressions in Disjunctive Normal Form (DNF).[7] The color names in these expressions serve as Boolean variables whose values are taken from the color bindings of the current thread.[8] Each variable is set to true if the current thread is associated with that color; it is set to false otherwise. The color constraint documents the permitted combinations of color bindings that may invoke the method. This annotation is often placed in the JavaDoc comment for the method.[9]

Example We return to our AWT/Swing GUI example to see why this is interesting. Recall that most AWT and Swing methods should be invoked only from the GUI's event thread. We write the AWT-thread-only constraint as "`@Color java.awt.AWT & !java.awt.Compute`". Figure 2.2 shows the annotation for method `java.awt.Button.setLabel`. The annotation states that `java.awt.Button.setLabel` must be invoked from the `AWT` thread; it also states that `Compute` threads are forbidden to invoke the method. We can also place color constraints on constructors, as seen in the figure.

Writing a color constraint provides benefit for the end user in three ways. First, the constraint documents the intended thread usage for the annotated method. Secondly, when assured by analysis, the color constraint provides confidence that the code inside a method will be invoked only from threads that satisfy the stated constraint. Thirdly, as additional constraints are added to the program the prototype assurance tool will ensure that the body of our annotated method does not violate some other method's color constraint by invoking the other method from the wrong thread role.

Global color constraints

Should a thread ever be permitted to be both an `AWT` thread and a `Compute` thread at the same time? The annotations we have seen so far do not answer this question. But the GUI thread usage policy strongly implies that `AWT` threads and `Compute` threads are distinct; no thread may be of both kinds simultaneously. We can model this property using the `@IncompatibleColors` annotation.

[7]Each DNF expression is a sentence made up of the disjunction of clauses. Each clause is the conjunction of terms. Each term is either a color name (a positive term) or a negated color name (a negative term). We use '&' as the Boolean AND operator, '|' as the Boolean OR operator, and '!' as the Boolean NOT operator.

[8]See Section 1.4.2 for a brief overview of color bindings, or Section 5.2.3 for a more formal definition.

[9]The annotation can also be placed in a stand-off file, but we're keeping things simple here.

```
package java.awt;
class Button extends Component implements ... {
  /** @Color java.awt.AWT */
  public Button(String label) {...}
  ...
  /** @Color java.awt.AWT */
  void setLabel(...) {...}
  ...
}
```

Figure 2.3: Code from `java.awt.Button` with improved constraint expression

@IncompatibleColors <color_name_list> This annotation establishes the global constraint that *at most one* of the named colors may be simultaneously associated with a given thread. As with the `@ColorDeclare` annotation, the `<color_name_list>` is a comma-separated list of color names. Because the `@IncompatibleColors` annotation is global in scope, it is best placed near the `@ColorDeclare` annotation for one or more of the colors in the `<color_name_list>`; this will typically be in a `package-info.java` file.

Example In the case of our GUI example, we would write

```
@IncompatibleColors AWT, Compute
```

As a consequence of this annotation we can reason that any thread known to be an `AWT` thread certainly is not a `Compute` thread, and *vice versa*.

By adding this `@IncompatibleColors` annotation to the declaration of the `AWT` and `Compute` colors we exclude the possibility that any thread could ever be associated with both colors simultaneously. This exclusion allows a simpler expression of our color constraint for `java.awt.Button.setLabel`, as seen in Figure 2.3. Note that because of the addition of the `@IncompatibleColors` annotation, the color constraint annotations in this figure have the same meaning as those in Figure 2.2; they are just simpler to write.

Global constraints provide end-user benefit by documenting the intent that certain thread roles should be distinct. The global constraints arising from this intent allow simpler expression of color constraints. The disjoint-ness property also permits expression of important models—in the case of our AWT/Swing example, it completes the AWT-only property needed to model the GUI thread usage policy.

2.3.4 Consistency Checking

Recall from above that the prototype assurance tool can use the color constraint on a method or constructor to check both whether it will be invoked only from threads that satisfy the stated constraint, and also whether the code inside the method does not violate some other method's color constraint by invoking the other method from the wrong thread role. At this point, it might be interesting to see how the prototype assurance tool reports findings of inconsistencies to the end user.

Example Returning to our AWT/Swing GUI example, we'll embark on analyzing GraphLayout, one of the standard example Java Applets. In the sections above we've seen some of the annotations needed to model the AWT/Swing thread usage policy, along with a color constraint for an AWT method and another for an AWT constructor.

What happens if we try analyzing GraphLayout now, with only these annotations and no further work? We get two errors. Specifically: "Color model not consistent with code at call to java.awt.Button.Button(java.lang.String) at Graph.java line 72." and "Color model not consistent with code at call to java.awt.Button.Button(java.lang.String) at Graph.java line 74."

To see why these errors arise, look at the code snippet in Figure 2.4 on the next page. On lines 72 and 74 we see two calls to the constructor for `java.awt.Button.Button(java.lang.String)`—these

```
66 public class Graph extends Applet implements ActionListener, ItemListener {
67
68   GraphPanel panel;
69
70   Panel controlPanel;
71
72   Button scramble = new Button("Scramble");
73
74   Button shake = new Button("Shake");
75
76   Checkbox stress = new Checkbox("Stress");
77
78   Checkbox random = new Checkbox("Random");
79   ...
```

Figure 2.4: Selected code from GraphLayout's `Graph.java`

are the locations of the reported errors. The problem we have found is that the constructor for Button requires that it be called from the `AWT` thread, but the calls shown in Figure 2.4 cannot be shown to originate from that thread. The true source of the problem is that we haven't written a color constraint for the constructor for class `Graph`. We'll see how to write the color constraint for that constructor in Section 2.3.11. This step of our example, however, raises the issue of how the prototype assurance tool handles code with partial or missing thread usage model information. We'll examine that question in the next section.

2.3.5 Unconstrained code

Methods that lack color constraints pose a problem for the thread coloring analysis. How should thread coloring analysis proceed when developer intent is missing? One approach is inference—used when the declared thread usage model in combination with the as-written code provides enough information to fill in the missing annotations. The other is a "no knowledge" approach, used when inference either fails or is not applicable. We'll start by considering the no knowledge approach.

No knowledge

What is the correct color constraint for code that lacks a thread usage model, whether expressed or inferred? Simple usability concerns suggest that the prototype assurance tool should be silent when presented with an entire program that lacks thread coloring annotations. The as-written code cannot violate a model that does not exist. Alternatively, when a program is fully annotated, the tool knows the thread usage model and should report errors when the code and model do not agree. This leaves us to consider what the thread coloring analysis should do for partially annotated code. There are four cases of interest, specifically the cross-product of code that is modeled (or not) invoking code that is not modeled (or is).

The cases where both caller and callee have an expressed thread usage model and where both lack an expressed thread usage model reduce to the behavior of fully-modeled or model-free, as above. To provide reasonable usability during the annotation process, we do not treat calls from modeled code to methods lacking a thread usage model as being in error—the constrained code cannot violate a constraint that is not present.[10] Conversely, when code that lacks a thread usage model invokes a method that has a color constraint, the prototype assurance tool *should* report an error. The designer of the annotated code requires that it be called only under certain conditions; the model-free code cannot show that it obeys the requirement. Table 2.1 on the facing page summarizes this behavior.

To achieve the desired behavior, the thread coloring analysis treats each method that lacks a color constraint as though its constraint were **`true`**. This choice produces exactly the behavior summarized in the table, and explains the errors seen in our running example at this point. It also explains why we did not see an error for the calls to the constructor for `java.awt.Checkbox` (lines 76 and 78 of Figure 2.3); because

[10]The code might still be wrong, of course. Remember that we are checking consistency of expressed thread usage model and as-written code; we are not establishing "correctness" in a more global sense.

	Callee has color constraint	Callee lacks color constraint
Caller has color constraint	Error check as usual	Always 'OK'
Caller lacks color constraint	Always in error	Always 'OK'

Table 2.1: Result of invocations between modeled and unmodeled methods

the constructor for class `Graph` is not yet annotated, its color constraint is **`true`** as is the color constraint for the constructor for the as-yet-un-annotated `java.awt.Checkbox`.

The end-user benefit for our "no knowledge" strategy arises entirely during the process of incrementally expressing the thread usage model for a program. The prototype assurance tool does not bother the user with error messages for code that is entirely lacking a thread usage model. It does, however, complain when unmodeled code attempts to invoke code whose thread usage model requires a particular thread color or colors. This alerts the end user that she may be violating the documented color constraints on the invoked code; she can resolve the question by extending her thread usage model.

Color inference

At this point, you may be asking yourself "Does this mean I'll have to annotate every method in my program? That would be way too much work!" Don't worry, you need not annotate everything in sight because the prototype assurance tool uses an inference algorithm to fill in annotations for you wherever possible.

The inference algorithm succeeds whenever it can see all possible invocations of a method, and there is a sufficient starting point for the inference. Let's look at these conditions separately. The prototype assurance tool can trivially see all invocations for `private` methods; they cannot be invoked outside their class. It can also see all invocations for methods that are not exported from their "home module".

Example Recall from Section 2.3.1 that we have placed all of GraphLayout into its own module, called `GraphLayoutApp`, and have not made any of its methods part of the module's public interface. Thus, `GraphLayoutApp` is the home module for everything in GraphLayout; none of the methods are exported from their home module, so inference can succeed—if it has a sufficient starting point.

The inference algorithm works forward from known color constraints, filling in unknown constraints as it goes. The resulting constraints are the Boolean-OR of the constraints of all invoking methods. Inference will succeed as long as all transitive callers either have an explicit constraint or have a successfully inferred constraint of their own.

But that's all a long-winded way of answering your real concern: "How do I know which methods should have color constraints?" The answer to this question is simple. If the method is visible to other modules, give it an explicit color constraint; inference will take care of almost all the remaining methods. We'll gloss over that "almost" for the moment.

The end-user benefit of color inference is a substantial reduction in the number of annotations to be written—as much as 12x in some cases.

2.3.6 Running example

Looking at GraphLayout's source code again, we see that we don't really know what to do next. We have a thread usage model for the AWT and Swing frameworks, and some information about the rules that their clients should follow, but we don't yet know how our client code plugs into the framework.

Consider the code snippet from `GraphPanel.java` shown in Figure 2.5 on the next page. On line 305 we see the `mouseDragged` method, which implements `java.awt.event.MouseMotionListener .mouseDragged`. We'll deal with the `mouseDragged` method in Section 2.3.7. Similarly, on line 306 we see a call to `e.getX()` and on line 308 we see a call to `repaint()`. What should the constraints on *those* methods be?

Think back to our earlier description of the thread usage policy for the AWT GUI framework; recall that it said that the vast majority of GUI methods may be invoked only from the `AWT` thread. The method `e. getX()` is exactly such a method. To choose a color constraint for `repaint()`, we take a close look at

```
  public void mouseDragged(MouseEvent e) {
    pick.x = e.getX();
    pick.y = e.getY();
    repaint();
    e.consume();
  }

  public void mouseMoved(MouseEvent e) {
  }
```

Figure 2.5: Code snippet from GraphLayout's class `GraphPanel`

the documentation for the AWT framework [4], where we learn that `repaint()` may be called from any thread.

At this point in the tour, you may ask "Does this mean that I also have to go annotate every method in all of AWT and Swing? No way!" Yes, *someone* has to go and annotate the API of AWT and Swing, but that someone need not be you.[11] In an ideal world the authors of a framework would provide the annotations for its thread usage model along with the code of the framework itself. This would be a good thing for the framework authors, because it would help their customers write working client code. But because thread coloring is a new idea, the authors of AWT and Swing have never heard of it; thus, I've written the annotations for those frameworks for you.[12]

You may also ask "Why doesn't color inference take care of this problem for me?" In theory, it could. But you'd have to put your code—the GraphLayout Applet, in this running example—in the same module as the AWT framework. More generally, you'd have to put your code in the same module as all of the code for all of the frameworks you use, along with all of the Java libraries and any third-party libraries you use. And you'd have to have source code for all of those frameworks and libraries, too[13]. Normal practice would be to put only related things in a single module. In most cases your code isn't part of the Java libraries, nor is it part of the AWT framework. And it certainly isn't part of that third-party library you're using. In short, those AWT and Swing methods come from a module that is not the home module of your code; rather, they are exported to you from their home module. As explained above, inference only operates within a single module, so it won't help here.

2.3.7 Color inheritance

Each method that overrides a method in a parent class or that implements an `Interface` method inherits the color constraint of its parent by default; this inheritance is transitive. The default inheritance of color constraints preserves substitutability; it also reduces yet again the number of annotations you must write. In fact, this default inheritance provides the color constraint annotations for the `mouseMoved` and `mouseDragged` methods seen in Figure 2.5. Because our local versions of the `mouseDragged` and `mouseMoved` methods implement the methods from `java.awt.event.MouseMotionListener`, they should have the same constraints as those found on the `Interface` methods.

Color inheritance is only a default, however. There are cases where a child method must have a color constraint that differs from that of its parent. Consider the `Runnable` interface, for example. It is trivial to write two different `run()` methods, one of which requires that it execute on the `AWT` thread and the other of which may *never* run on the `AWT` thread—see Figure 2.6 on the next page. The first `run()` method must always execute on the `AWT` thread, because it invokes an `AWT`-only constructor. The second `run()` method should never execute on the `AWT` thread, because its infinite loop would freeze the GUI, rendering it useless to the user. These two methods are substitutable in terms of the Java type system; they are not substitutable either in terms of their operation or in terms of what thread roles may execute them.

[11]And if you *did* have to annotate the entire framework it certainly would be far more work than annotating our little example Applet. In fact, it would be more work than is worth doing just for GraphLayout.

[12]Actually, I've written most of the annotations; some code in each framework remains to be annotated.

[13]This is not a fundamental requirement, but rather a limitation of the prototype assurance tool. To the extent that Java byte-codes are nearly isomorphic with the original Java source code, it should be possible to perform the necessary analysis on byte-codes.

```
/** an AWT-only run() method */
public void run() {
   ...
   Button scramble = new Button("Scramble");
}
...
/** A run() method that must never run on the AWT thread */
public void run() {
   while (true) {
      // loop forever
   }
}
```

Figure 2.6: Two incompatible `Runnable.run()` implementations

To accommodate non-substitutable cases like this, the prototype assurance tool permits an explicit color constraint on a child method that is not the same as that on its parent method. The tool issues a warning and continues with its analysis.

Color inheritance simplifies the problem of writing framework client code. In most cases the end user need not write explicit color constraints for client methods that override framework methods; rather, the client methods inherit their color constraint from the framework method. This significantly reduces the number of color constraint annotations the end user must write.

2.3.8 Running example

We return once again to the GraphLayout Applet. Now that we have all of the needed thread coloring annotations in the AWT framework, we re-run the prototype assurance tool to see what messages we get. Because we have not yet added any annotations to GraphLayout itself, we are not surprised to find that we still get the error messages we originally saw in Section 2.3.4 on page 21. In fact, we see even more messages indicating that the constructor for class `Graph` is invoking `AWT`-only methods, but is not itself colored. Once we deal with that minor problem, we see that most of GraphLayout gets its color constraints from the AWT framework by inheritance. The only methods remaining un-colored are `GraphPanel.run()` and `GraphPanel.relax()`. In Figure 2.7 on the next page we see excerpts of class `GraphPanel`. The `run()` method is invoked by Java library code after the `Compute` thread is started. It repeatedly calls the `relax()` method, does some additional work, and then sleeps to reduce the change rate of the graph display. The `relax()` method uses a relaxation algorithm to move the nodes around, and then calls `repaint()` to tell the GUI that the graph has changed and should be redrawn.

The `run()` method in `GraphPanel` is invoked as the main entry point of a `Compute` thread in accordance with the rules of the Java library's `Thread` and `Runnable` frameworks. We note this by annotating the method with a matching color constraint (line 1). We avoid annotating the `relax()` method at all (line 19), because it is neither part of the public interface of the `GraphLayoutApp` module, nor is it an entry-point such as `run()`. Instead, the prototype assurance tool's inference algorithm computes a suitable color constraint for `relax()`.

We are nearly done analyzing GraphLayout. At this point, all of the code in GraphLayout itself has either an explicit color constraint, an inherited color constraint, or an inferred color constraint. All that is missing is the annotation for the constructor for class `Graph`. In the next sections we will see an alternate approach to annotating the `run()` method in `GraphPanel` and how to apply a color constraint to an implicit constructor.

2.3.9 Granting colors

There is a reasonable alternate approach to annotating the `run()` method in `GraphPanel`. We might choose to "grant" the `Compute` color inside the method rather than requiring that its caller already "know" that it is a `Compute` thread.

```
/** @Color Compute; */
  public void run() {
    Thread me = Thread.currentThread();
    while (relaxer == me) {
      relax();
      if (...) {
        Node n = nodes[...];
        if (!n.fixed) {n.x += ...; }
      ...
      }

      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {
        break;
      }
    }}

  synchronized void relax() {
    for (int i = 0; i < nedges; i++) {
      Edge e = edges[i];
      ... = nodes[e.to].x - ...;
      ...
      nodes[e.to].dx += ...;
      ...
    }
    ...
    repaint();
  }
```

Figure 2.7: Code snippet from `GraphPanel.java` showing `run()` and `relax()` methods

```
/** @Color !AWT */
public void run() {
  //@Grant Compute
  Thread me = Thread.currentThread();
  while (relaxer == me) {
    relax();
    if (...) {
      Node n = nodes[...];
      if (!n.fixed) {n.x += ...; }
    ...
    }

    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      break;
    }
  }
}
```

Figure 2.8: Alternate annotation of `GraphPanel.run()`

@Grant <color_name_list> This annotation establishes a binding between any thread that executes the method and the listed colors. The annotation may appear only at the beginning of a block. The binding expires on outwards block exit, and so is removed on exit from the method. As with the `@ColorDeclare` annotation, the `<color_name_list>` is a comma-separated list of color names. Attempting to grant colors that would violate a global constraint is an error.

Example The color constraint at the beginning of the `run()` method in Figure 2.8—`@Color !AWT` —serves to establish the required pre-condition for granting the `Compute` color. Recall that we have required that no thread may ever be both `AWT` and `Compute` at the same time. If some future programmer were to invoke this `run()` method from the `AWT` event thread, our `@Grant` annotation would bind the `Compute` color to the event thread. This would violate our global constraint, so we must prevent it from happening by requiring the "no `AWT` thread may ever get here" constraint.

There is a subtle difference between the two annotation approaches. The "just use a color constraint" approach requires that the calling method already know both that it is *not* an `AWT` thread, *and* that it is a `Compute` thread. The "`@Grant` inside the method" approach requires only that the calling method know that it is *not* an `AWT` thread; the fact that it executes the `run()` method will *establish* that it is a `Compute` thread, at least until it returns from the `run()` method. It should be clear that the color environment inside the `run()` method is identical in both cases; the difference is the constraint placed on callers.

To understand this distinction more clearly, recall that threads can have many colors simultaneously. Suppose that a hypothetical method whose color environment is `!AWT & BackgroundRendering` contains a call to `GraphPanel.run()`. The "just use a color constraint" approach would cause the prototype analysis tool to report an inconsistency at this call site, because the calling color environment does not have the `Compute` color. By comparison, the "`@Grant` inside the method" approach would cause the prototype analysis tool to report that the call is OK, because `GraphPanel.run()` does not require the `Compute` color. Choosing between these approaches requires deciding whether or not `(AWT | Compute)=` **`true`**; that is, whether or not it is possible to have a thread that is neither `AWT` nor `Compute`. Either choice is potentially correct, depending on your desired semantics.

The `@Grant` annotation allows a programmer who knows that execution at a certain program point is possible only when the current thread is actually of a particular kind to "stamp" the current thread with the appropriate color. The benefit is that the programmer can establish an initial color context at an appropriate program point.[14]

2.3.10 Scoped promises

To reduce the need to explicitly annotate every `public` method, we can use a pattern-matching technique that applies one annotation to many methods. This technique relies on the fact that programmers use highly-stylized naming conventions. For example, it is usually reasonable to guess that methods whose names begin with "is," "has," or "get" probably read fields from their class and that methods whose names begin with "set" or "modify" probably write fields in their class. This tendency towards stylized naming lets us use lexical name matching and type matching to identify those Java entities in a scope for which we intend to place a particular annotation.

This pattern-matching technique is called a "scoped promise;" it has some important generalizations (see Section 2.4). The basic idea is that, for every Java entity in scope that matches the pattern, the annotation is implicitly applied at that location. The general syntax is:

`@Promise "payload_annotation" for location_specifier` The `payload_annotation` is the annotation to be applied and `location_specifier` is the specification of the Java entities to which the annotation will be applied.[15] The Fluid project—of which this research is a component—supports many kinds of annotations; a scoped promise can also specify other Fluid annotations as its payload. The `location_specifier` supports AND, OR, and NOT operations on groups of wild-carded names. The wild-card characters are the familiar '*' and '.' from Unix shell expressions. The "scoped" part of "scoped promise" arises from the fact that the `location_specifier` is not applied globally; rather, the matching of Java entities takes place only in the lexical scope containing the scoped promise.[16]

When matching methods and constructors[17], the `location_specifier` includes both a name-match component and a parameter-match component. The name-match component uses the sequence "*" to match any sequence of characters, and "." to match any single character. The `location_specifier` "is*" would match any method name that starts with 'is'. The `location_specifier` "*Reset" would match any method that ends in "Reset"; and "Set*Value" would match any method that started with "Set" and ended in "Value". To simply name all methods, the shorthand notation "*" matches all method names. To match a particular method signature, the programmer provides names, separated by commas—one for each parameter, with wild-cards as needed. The special wild-card "**" matches any number of parameters of any type. Thus a `location_specifier` for all methods with any number of parameters would be written "*(**)", and a `location_specifier` for all constructors with any number of parameters would be written "new(**)". To match all constructors and all methods in a scope, regardless of signature, the programmer uses the `location_specifier` "**(**)".

For example, the scoped promise

```
@Promise "@Color␣(Reader␣|␣Writer)" for (is*(**) | has*(**)) & !(hasGlobal*(**))
```

matches every method in scope whose name starts with "is" or "has", *except* those whose name starts with "hasGlobal." The "**" for the argument types indicates that we intend to match all methods regardless of the number or type of their parameters.

I've introduced scoped promises here in order to show how to use them to annotate an implicit constructor without making the constructor explicit. This is an obvious benefit of scoped promises. They also provide a more important benefit—careful use of scoped promises can reduce the number of annotations the end user

[14]The similar `@Revoke` annotation allows the programmer to remove a color from the current thread.

[15]I did not invent scoped promises; they are a result of Halloran's thesis research. Thread coloring is, however, a heavy user of scoped promises.

[16]The term "lexical scope" is used rather loosely here. Within individual compilation units, the term has exactly its ordinary meaning. However, we also support matching over entire packages for scoped promises placed in the `package-info.java` file; support for module-scoped matching is planned. These extensions to the ordinary idea of lexical scope are in the style of Aspect-oriented Programming's point-cuts; they pose no additional challenges beyond the engineering realm.

[17]The full syntax for scoped promises supports matching variables, types, and many other entities not relevant to this discussion. See [25] for a more complete treatment of the subject.

```
/**
 * @Promise "@Color java.awt.AWT" for new()
 * ...
 */
public class Graph extends Applet implements ActionListener, ...
{
```

Figure 2.9: Scoped promise applied to class Graph

must write by an order of magnitude or more. I'll discuss this latter benefit at greater length in Section 2.4 on page 33 and in Section 3.5.2 on page 62.

2.3.11 Running Example

Returning to our running example, we now consider how to write the color constraint for class `Graph` 's constructor. Writing the color constraint's expression is trivial; it is `@Color java.awt.AWT`. Our problem is where to place the annotation for a constructor that has no explicit presence in the code. We have two alternatives. Firstly, we could simply add an explicit constructor and annotate that. Alternatively, we can use a scoped promise to apply the annotation directly to the implicit constructor. We'll use the scoped promise this time.

In this case, we wish to place our desired color constraint on the implicit constructor. We place the scoped promise on class `Graph` as seen in Figure 2.9. The "`for new()`" part of the scoped promise is the `location_specifier`. In this case "`new()`" indicates that the promise applies to the implicit no-arguments constructor in the current scope; that is, to the implicit constructor for class `Graph`. We might reasonably have chosen the `location_specifier` "`new(**)`" to document a commitment that all constructors for this class should have the color constraint `@Color java.awt.AWT`. This would have the identical effect on the code as currently written; by matching all possible constructors we would indicate our expectation that even as-yet-unwritten constructors must obey this constraint.

2.3.12 Issues previously glossed-over

We now consider two issues we have previously glossed over.

No color constraint

Certain methods are intended to operate correctly no matter what thread invokes them. Recall from Section 2.3.6 on page 23 that the AWT `repaint()` method is such a method. The methods in class `java.lang.Math` are another fine example; computing a cosine or an absolute-value does not depend on any special knowledge or properties of the invoking thread. We can model this with the `@Transparent` annotation.[18]

`@Transparent` Transparent methods may be freely invoked from any thread or thread role. Transparency places a stringent restriction on the implementation of a method so annotated: it may not invoke any method or access any data region that has a non-transparent color constraint. This is a consequence of our lack of knowledge about which kind of thread might invoke the transparent method. Because the transparent method might be invoked from any arbitrary thread role, it is unable to satisfy any constraint on a method it wishes to invoke.

Although the `@Transparent` annotation represents the same color constraint we have for "no knowledge" methods, it serves quite a different purpose. A method annotated as being `@Transparent` has been analyzed by its implementor and deliberately marked as having no color constraint; it is *supposed* to be unconstrained. By comparison, a "no knowledge" method may simply not yet have been considered. Thus,

[18]Note that `@Transparent` is semantically equivalent to `@Color` **`true`**, that is the weakest possible constraint.

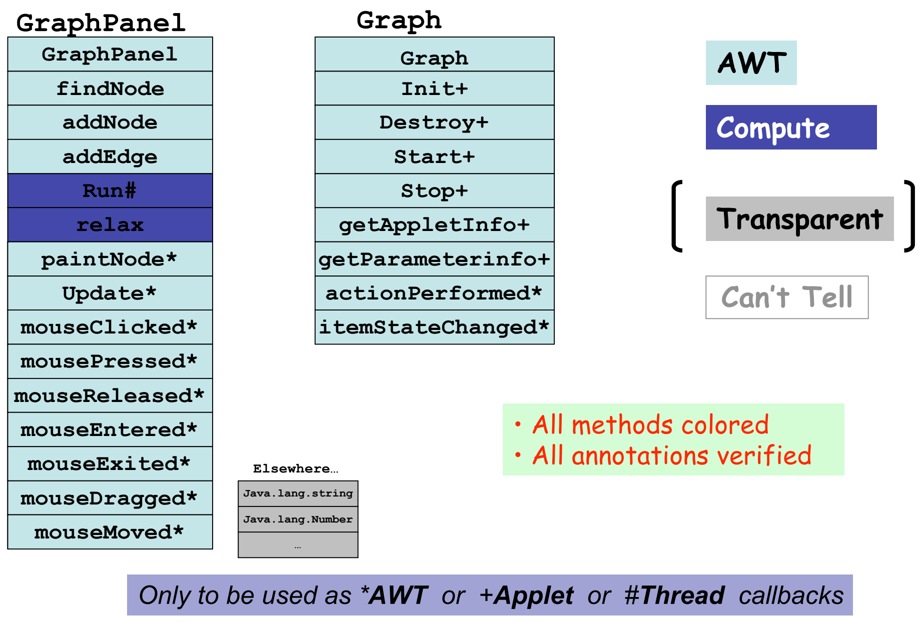


Figure 2.10: Thread coloring results for GraphLayout

`@Transparent` represents a semantic commitment by the method's author or reviewer. It also serves to document that commitment for other developers.

Importing color declarations

We have been using fully-qualified color names throughout this tour. However, it seems painful to write `@Color java.awt.AWT` all the time. Wouldn't it be easier to write `@Color AWT` instead? We can do this by importing color declarations from an external class or package with the `@ColorImport` annotation.

@ColorImport <java_place> The <java_place> specifier has the same syntax as ordinary Java imports. This annotation is similar in intent to Java's import clause; it makes the color declarations in the named place directly visible in the scope it annotates. Typical placement for this annotation is at the beginning of the outermost class in a compilation unit.

2.3.13 Running example

To use short names in all of our color constraint annotations in GraphLayout, we would simply add the annotation `@ColorImport java.awt.*` to each compilation unit in the Applet, thus making the declarations of the `AWT` and `Compute` colors directly visible in those CUs. [19]

We have now finished coloring the code in GraphLayout. Figure 2.10 shows the resulting colorings. Note that a number of library methods are colored "transparent;" this is the result of additional modeling of the Java libraries. As with the AWT and Swing frameworks, I have written thread coloring annotations for parts of the Java libraries.

[19]The rather unfortunate ".*" part of the syntax is a result of re-using the syntax for Java's demand-import clause. This is an artifact of the prototype assurance tool's implementation; it is not inherent in the analysis.

Figure 2.10 also documents the various frameworks that GraphLayout's methods participate in: AWT, Applet and Thread. These are indicated with '*', '+', or '#' as documented in the text bar at the bottom of the diagram.

2.3.14 Coloring data

In the earlier parts of our tour we have seen how to apply thread colors to GraphLayout's code. "What," you might ask, "can we do for data?" The prototype assurance tool offers two options. You can ask it to compute which thread colors access a particular data region, or the tool can enforce a color constraint that you have applied to a particular data region.

"What good is that?" you may ask.

If you know enough about the mappings between colors and threads, you can use this information to decide which data regions are potentially shared between threads. Data regions that are potentially shared between threads require some form of synchronization for correct program operation. You can also use data coloring to restrict a data region so that it may be accessed only from one thread color. If you know that there is only a single thread that has that color, you know that the color-constrained data region is actually thread-local. Because thread-local data is touched from only one thread, it does not need any form of synchronization; you can simplify your code by removing all locks from these regions. Lastly, when a data region has a color constraint, the prototype assurance tool will ensure that code written later does not violate that constraint. For the thread-local case, this maintains the condition that made it safe to remove the synchronization.

To show how to use these features, we will first briefly discuss data regions. We'll then move on to a discussion of thread color multiplicity before finally discussing data coloring itself.

Data regions

The idea of data regions comes from earlier research by members of the Fluid group [24]; support for thread coloring of data (a.k.a. data coloring) builds on this previous work. Data regions are identified areas of memory that contain objects or portions of objects. We'll use annotations to establish the mapping of program objects into data regions. Because these annotations are part of earlier work, we will not cover them in detail; for a more complete explanation, see [25].

```
/**
 * @Region TheGraph
 * @MapFields nnodes, nodes, nedges, edges into TheGraph
 *
 * @ColorizedRegion TheGraph
 */
class GraphPanel extends Panel implements Runnable, MouseListener ... {
  ...
}
```

Figure 2.11: Data coloring annotations for `GraphPanel`

```
class Node {
  //@MapInto GraphPanel#TheGraph
  double x;
...
```

Figure 2.12: Data region annotation for `Node`

Example Figures 2.11 and 2.12 show example data region annotations for part of GraphLayout. The `@Region TheGraph` annotation declares the existence of a data region named `TheGraph`. We intend that all of the data for the GraphLayout's graph should be contained in that region. To accomplish this, the `@MapFields nnodes, nodes,...` annotation notes that the named fields of all instances of class

`GraphPanel` are all part of the data region `TheGraph`. The `@MapInto` annotation in Figure 2.12 has a similar effect for the `x` field of all `Nodes`. Greenhouse–Boyland object-oriented effects analysis [24] determines, for every data reference in the program, which regions may be read or written by that data reference.

There are many uses for the Greenhouse–Boyland effects analysis; the only one that matters here is that it provides the basis for coloring data. The end-user benefit of data region-related annotations (as related to thread colors) is primarily that they enable the prototype assurance tool to answer questions about threads and data regions.

Thread color multiplicity

To reason about the meaning of data coloring results, the end user must know something about the maximum number of threads that might be bound to a particular color at once. The bad news is that computing this runtime property at compile time is equivalent to solving the halting problem. The good news is that the tool does not need to do so. Instead, thread color multiplicity is a purely declarative attribute; the prototype assurance tool assumes that the annotation is correct.

`@MaxColorCount <color_name>, {0 | 1 | n}` This annotation declares the maximum multiplicity of the named thread color. It is purely declarative, and is not checked in any way. The construct `{0 | 1 | n}` indicates that any one of “0”, “1”, or “n” is a valid choice. Because this annotation has global effect, it should be placed near the declaration of the named thread color; this is typically in a `package-info.java` file.

The interesting maximum multiplicity numbers for our purposes are zero, one, and many. If no actual thread can be bound to a particular color at runtime, any code with that color should be unreachable.[20] If the multiplicity for a thread is one, any data region accessed only from that color and from no other clearly has single-threaded access;[21] we can reason that such data need not be locked. Finally, if the data is accessed from one or more colors with multiplicity greater than one (or with unknown multiplicity), we can reason that the data may be shared between threads and so requires synchronization.

Example In the case of our running example, the annotation indicating that there is at most one AWT event thread at a time is written

```
@MaxColorCount AWT, 1
```

This annotation is included with the other color annotations for the AWT and Swing frameworks.

The prototype assurance tool makes no locking recommendations, and attempts no reasoning based on the multiplicity of thread colors. It permits end users to use the `@MaxColorCount` annotation to document their expected color multiplicities. Reasoning about sharing properties is left to the end user.

Computing data access colors

It is often useful to ask the prototype assurance tool to compute the accessing color environment for a data region. We use the `@ColorizedRegion` annotation for this purpose.

@ColorizedRegion <regionName> This annotation tells the prototype assurance tool to compute the color environment from which the named region is accessed. The result will be the union of all accessing color environments; that is, it tells you all color environments from which the named region may be accessed on any execution of the portion of the program being analyzed. This annotation should be placed near the declaration of the named region.

[20]The prototype assurance tool does not take advantage of this property.
[21]There are more complicated checks for single-threaded access; this one is simple to explain.

```
/**
 * @Region Edges
 * @MapFields from, to into Edges
 * @ColorConstrainedRegions (AWT) for Edges
 */
class Edge {
   Node from, to;
   ...
```

Figure 2.13: Data coloring annotations for `Edge`

Example The “`@ColorizedRegion TheGraph`” annotation in Figure 2.11 on page 31 tells the prototype assurance tool to compute the color environment from which the region `TheGraph` is accessed. The resulting information allows you to reason about whether or not your data region may be shared between threads at runtime. In this particular case, `TheGraph` is touched both by the `Compute` thread (while performing the relaxation algorithm) and also by the `AWT` thread (for display of the graph). Because we know that these threads are distinct, we can conclude that `TheGraph` is in fact shared between threads.

The end-user benefit is obvious. Shared data should have some form of consistency control (likely locking) and is a candidate for a locking analysis. Unshared data need not be locked.

Color constraints for data

Let us suppose now that we decide that the edges of the graph should be touched only from the `AWT` thread. We use the `@ColorConstrainedRegions (AWT)` **`for`** `Edges` annotation for this purpose.

`@ColorConstrainedRegions <constraint_expression> for <regionNames>` The constraint expression is a Boolean expression over color names; its syntax is identical to the `<constraint_expression>` for the `@Color` annotation. The `<regionNames>` is a comma-separated list of one or more data region names. This annotation is for data what the `@Color` annotation is for code; that is, it declares that any access to the named data region must come from a color environment that satisfies the constraint expression.

Example Figure 2.13 shows the data region and thread coloring annotations for class `Edge`. The `@Region` and `@MapFields` annotations operate as before, except that this time we are mapping into the data region `Edges`. The “`@ColorConstrainedRegions (AWT)` **`for`** `Edges`” annotation establishes the color constraint `AWT` for the named regions (only one in this case).

Because the edges of the graph are not accessed during the relaxation algorithm, the prototype assurance tool finds no errors during its checking; each access to the edges of the graph does in fact come from the `AWT` thread. Further, because we have declared that there is at most one `AWT` event thread at a time, and because all accesses to the edges of the graph come only from the `AWT` color, we can conclude that the edges of the graph are entirely thread local. If synchronization was a concern in the GraphLayout example, we would be justified in removing any synchronization for the edges in the graph.

The end-user benefit of this annotation should be clear; it allows the programmer to constrain his program so that only the intended thread roles can access the annotated data region. If, as in our example, the constraint limits access to a single thread, the programmer is guaranteed that he does not need to synchronize access to this data region in a fully analyzed program.

2.4 Design intent at scale

Our thread usage model could be large for a variety of reasons: a very large program has many places to annotate; a program with many thread roles will complicate our expression of color constraints; and a complex thread usage model is fundamentally larger to express. Each variety of scale presents us with problems to solve. A common theme of my approach to these problems is to localize the complexity so that the end user need wrestle with it only once; the prototype assurance tool then uses inheritance and inference to spread the annotations from the complex portion of the model around the rest of the program.

2.4.1 Scaling design intent for large programs

At this point in the guided tour, some end users are probably saying: “That’s all very nice, but you haven’t tried something like my multi-million-line program. How are you going to handle that?”

The basic issue with large programs is a direct result of their size—there’s an awful lot of program to annotate. Annotating millions of lines of code could be a large effort, even with very concise annotations. Fortunately, it’s not as hard as it may first appear. Let’s take a look at how you can reduce model expression effort by using the features we saw during the guided tour.

Incremental adoption model You don’t have to annotate the entire program before seeing benefit from your work. The Fluid group’s analyses and tooling, including thread coloring and modules, are designed for incremental adoption. Our idea is that each small increment of model expression effort should provide an increment of immediate benefit. The thread coloring and module analyses take several approaches to achieve this policy. First, the prototype assurance tool doesn’t bother you with error reports about code that lacks thread usage model information. Such code has no reported errors, and receives no assurances. The prototype assurance tool reports errors only when as-written code is not consistent with expressed programmer intent. Section 2.3.12 on page 29 discusses this approach in more detail. Secondly, each method or data region that has a color constraint (whether programmer-written, inherited, or inferred) receives an assurance that the constraint will be obeyed by all checked code. Adding such a constraint expands your island of assurance immediately. Of course, it may also introduce error messages in code that lacks a thread usage model, but that’s a *good* thing—the prototype assurance tool cannot assure that un-modeled code obeys the constraint that you just introduced.

Scoped promises The primary purpose of scoped promises is to support concise expression of design intent that is then spread across all matching program points. The matching and spreading is done by the prototype assurance tool, without additional effort on your part. Scoped promises can provide a one or two order of magnitude decrease in the number of annotations you need to write. You can read more about scoped promises in Section 2.3.10 on page 28; Section 3.5.2 on page 62 includes an investigation of their effect on numbers of annotations.

Color constraint inheritance Methods that override superclass methods or implement `interface` methods inherit the color constraint of their parent by default. Thus, by annotating a method near the root of the inheritance tree, you effectively write the color constraint for all of its children too. Even better, when the parent method is part of a Java library or other framework, the library’s authors may well have written the constraint for you.

Module-at-a-time analysis The prototype assurance tool operates on one module at a time. You can carve off a module of interest to you and start annotating inside it. You need not annotate the rest of the program to get started. Rather, you will wind up annotating those public interfaces that are provided by the rest of the program and are used within your module. Thus, you can focus your annotation work on the subset of the program you are most concerned with. In addition, during analysis you need not wait while the tool analyzes portions of the program that you don’t care about; rather, it only spends time analyzing the modules containing code of interest to you.

Color constraint inference As we saw in Section 2.3.5 on page 23, the prototype assurance tool can use the annotations on module interfaces to infer most of the color constraints for methods that are not exported from their home module. This inference reduces yet again the number of annotations you need to write.

These techniques are quite effective in combination. Section 3.5.2 on page 62 presents a case study on how many annotations I needed to module-ize and model the thread usage policy of a 140KSLOC program. The current prototype assurance tool requires 6.3 annotations per KSLOC; I believe this can be reduced another order of magnitude to about 4 annotations per 10KSLOC.

2.4.2 Scaling design intent for many thread roles

Many thread roles in the same program raises two different problems. First, when the thread roles are distinct, any time we wish to permit one of them we must exclude the others. As we have seen in Section 2.3.3, this is easily handled with the `@IncompatibleColors` annotation. You express the incompatibility once; the prototype assurance tool ensures that it is properly considered at every color constraint, with no further action on your part.

The second problem is that large numbers of colors sometimes means that there are many distinct cases to describe in a color constraint expression. Although the full cross-product is not common, you might need to write a Boolean expression that has a dozen or more clauses in disjunctive normal form. This does not change the number of annotations to write, but it can make some individual annotations complex and tedious to write. In this case you can save work by using the `@ColorRenames` annotation.

`@ColorRenames <name> for <colorExpression>` The `@ColorRenames` annotation provides a very simple macro capability. Anywhere this renaming annotation is in scope, you can write the given name; the tool will automatically expand it into the given color expression. The `<name>` can be any simple name. The `<colorExpression>` is a Boolean expression over color names whose syntax is identical to that of the `@Color` annotation. Typical usage is to place these annotations near the declaration of one of the color names used in the `<colorExpression>`, typically in a `package-info.java` file.

Example Consider, for example, the annotation

```
@ColorRenames BigExp for ((Big & !Short) | (Expression & With & !Few) |
    (Clauses & That & Would & Be & !Interesting) | (To & Write & Very &
    !Few & Times))
```

Anywhere this renaming annotation is in scope, you can use `BigExp` in place of the long, tedious expression.

It's a good idea to place renaming annotations near the declaration of the color names they use. If such placement is not possible, the `package-info.java` file for one of the packages in which the rename will be used is another good location. Color renamings also improve the quality of error reporting by providing short names for long expressions; see Section 6.6.1 on page 161 for more information.

2.4.3 Fundamentally complex threading models

Some thread usage models are large because the underlying threading is itself complex. Sections 4.4 on page 84 and 4.8 on page 112 describe two applications that have complex threading models. My basic approach to tame this complexity is to understand what the program is doing and to express the thread usage policy by combining annotations as needed in a few places. This allows the tool's inference and inheritance support spread the results of this complexity throughout the program as needed. Other than the inference and inheritance support, this approach is more a matter of attitude than of tooling. No case study to date has yet presented a threading model so complex as to make this approach insufficient.

Chapter 3

Modules

3.1 Introduction

This chapter discusses the Fluid module system. Its particular features enable composable module-at-a-time analyses, thus permitting scaling to large programs. The module system also enables a "module-at-a-time" annotation inference algorithm, providing a roughly 12x reduction in annotation count for thread coloring. This leads to a much lower number of annotations required to perform thread coloring for a 140KSLOC application: 6.3 annotations per KSLOC with the prototype implementation, reducible to 0.25 annotations per KSLOC with modest additional engineering effort.

The chapter begins by motivating the use of modules, for both software developers and analysis writers, then presents the criteria that drove our specific design choices. It continues with a discussion of our detailed design decisions, complete with the annotations that express the module-ization of code along with examples of their use. Section 3.5 presents some examples of analyses that take advantage of modules and one large case study demonstrating the benefits of combining modules and inference to reduce the number of annotations users must write. The chapter then compares the Fluid module system with a number of prior module systems, and concludes with a discussion of future work.

Because modules are not a new idea, the discussion of the Fluid module system is limited to a single chapter. Discussion of the details of the Fluid module system begins in Section 3.4.

3.2 Why use modules?

We focus on two constituencies for the use of modules: software developers (broadly construed to include designers, architects and managers) and analysis developers (including compiler writers, static source code analysis builders and so on). In this section, we give a basic introduction to modules and discuss reasons why these two groups should be interested in using modules to support their work.

Readers who are familiar with modules may wish to skip to Section 3.3 on page 40, which discusses why we chose to design a new module system.

3.2.1 Software developers and managers

Change

For software developers, modules serve as a mechanism to manage the potential impact of future changes—both anticipated and unanticipated—on their system. Parnas introduced systematic thinking about the issues and criteria behind "modularization" in [46]. He recommended abandoning control flow as the main criterion, and proposed information hiding in its place. He defined information hiding as "every module [...] is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings." Many mechanisms have been developed in support of this goal—including abstract data types, referential transparency, and object-oriented programming with its inheritance and substitutability, to name just a few—but perhaps the most broadly applicable of all has been control of compile-time visibility. Nearly all popular programming languages—such as C, C++, C#, Java, Ada, and many others—provide fine-grained control over visibility, at least to the degree of hiding "private" portions of the implementation of compilation units[1] (CUs).

Concurrent development is another mechanism for coping with change. Developers working on distinct portions of a large program should be able to do so with minimal interference with, or impact on, each other's work. Separate compilation addresses this goal (among others), by allowing developers to make non-interface changes without impact on the work of others. This is clearly an aspect of dealing with change; if programs never changed, further development would not be needed and the need for concurrent development would be greatly reduced (although not eliminated, as we still desire concurrency in the original development process).

Control of coupling and cohesion is another driver for the use of modules. Coupling has been defined as "the extent to which a software part is related to other software parts [...] a property of an individual software part, or more specifically a relation between an individual software part and its associated software system" [8]. Cohesion is "the degree to which data declarations and subroutines of a module[2] are conceptually related, based on information known at the end of high-level design [8]." Controlling coupling assists developers in coping with change by helping them to limit the scope of changes. Unwanted coupling can cause necessary changes to ripple outwards through a program. Traditional module systems and languages often provide good support for control of unwanted coupling at the level of compilation units. For example, Ada [52]—in both its current (Ada05) and prior (Ada95) incarnations—provides a hierarchical package system that allows developers to distinguish, depend on, and extend both the public and private parts of their compilation units.

Scale

As programs grow, their increased scale adds to the developer's challenges. In large scale programs it may be desirable to reduce the potential for coupling by hiding not only the bodies of compilation units, but also their interfaces. Some current module systems (such as Java's package system and C#'s namespaces) provide one level of hiding as part of a flat system; no currently popular system provides native support for either a hierarchy or a layered system. A richer system for controlling compile-time visibility could help designers and developers to both express and enforce visibility limitations, and so force compliance with intended limitations on coupling.

When Parnas introduced the idea of using modules for information hiding, the modules he discussed were roughly the scale of today's compilation units. His suggested criteria are precursors of current measures of cohesion and coupling. His modules are notably smaller than those of most current systems—in fact, one might say that today's modules are typically comprised of many units, each of which could be a Parnas module. That said, the goal of information hiding and control of both coupling and the potential for coupling is a key component of both approaches.

Software architecture patterns help developers to design programs whose shape and degree of coupling are appropriate to the task at hand. Two widely used architectural patterns for system building identified in [53] are hierarchical composition and layering. Shaw defines the first as being based on a "call and definition hierarchy, subsystems often defined via modularity" with components that are "procedures and explicitly visible data." Layered systems are described as being "suitable for applications that involve distinct classes of services that can be arranged hierarchically," adding that "frequently, each class of service is assigned to a layer[...]." The system model is usually a "hierarchy of opaque layers." Halloran's system of layers [29] provides the Fluid group's implementation of layering. The Fluid module system provides direct support for building systems via hierarchical composition.

In a hierarchical system, developers compose the larger system out of subsystems, each of which has a defined and deliberately limited interface. The subsystems are composed out of sub-subsystems, again with defined and limited interfaces, and so on. Further, the interface a subsystem presents to its peers is typically smaller than the union of the interfaces of the sub-subsystems that it contains. Note that a sub-subsystem, in this context, may itself be a medium-to-large body of software possibly containing more than 100KSLOC. The sub-subsystem may itself be organized in some combination of modules and layers (or other architectural patterns); the picture is entirely recursive. Supporting this hierarchical style of system building was another concern in the design of the Fluid module system.

[1]Note that compilation units are roughly the scale of module considered by Parnas in [46].
[2]Note that 'module' in this context refers to a single compilation unit.

A successful module system should support multi-level hiding of interfaces to assist in reducing coupling, but it should not over-constrain the structure of the resulting system. Module systems that are strictly flat do not over-constrain their clients, but provide only one level of hiding. They thus fall short in reducing coupling. Module systems that support arbitrary nesting (including multiple parents) provide multi-level hiding with minimum constraint. However, unless executed with care, they allow confusing combinations of visibility. Tree-structured module systems make a good match with real-world systems, and provide excellent support for multi-level hiding of interfaces and control of coupling. Strict tree-structure, however, is sometimes too strong a constraint.

Source code management

Source code management systems have long supported nesting and hierarchy. The Concurrent Versions System (CVS) [28] supports a hierarchy of nested modules[3] within a source code repository. The main effect of this nesting is that checking out a parent module checks out the child as well. Similar capabilities are found in many other source code management systems, including Subversion [58], ClearCase [50] and others. Note, however, that module hierarchies in source code management systems generally have no direct impact on programming language visibility.

3.2.2 Analysis developers

Analysis developers have not typically been concerned with module systems beyond the scope of compilation units. There are many examples of analyses that operate on individual compilation units—the entire compiler optimization literature springs to mind. This approach to analysis benefits from the typical programming language's support for information hiding and static visibility control. Its major limitation is the inability of this approach to apply analysis algorithms whose concerns span many compilation units. Yet program analysis tools often operate over larger chunks of a software system than individual compilation units, and may benefit from the use of, for example, inference-based analyses over large swaths of code.

One common response to this restriction has been to analyze entire programs. Whole-program analysis appears attractive; it allows the analysis to be certain that it has seen all the code in the program. The drawbacks are:

- For any given analysis, there is some program that will exceed any reasonable degree of computational resources; for example, some users may wish to analyze multi-million SLOC programs. This problem is exacerbated by the worse-than-linear performance of many analysis algorithms.
- Source code for the entire program may be unavailable. Consider, for example, the problem of analyzing the implementation of a software framework. It is quite conceivable that client code is as yet unwritten at the time of analysis. Even if some client code is available, other client code remains to be written (otherwise there would be no point to the framework).
- Even when both of the above issues are surmounted, users may find that analysis of an entire program requires too much time. They may well be better served by partial analysis, as long as its results can be successfully combined with analyses of other portions of their program.

An intermediate chunk size would enable program analysis tools to operate on usefully large pieces of the program without moving to whole-program analysis. My personal position is that modules and their boundaries are a useful intermediate chunk size. I am not alone in this belief. Systems such as MJ [10] support per-module analysis. The Ada specification [52] explicitly permits, but does not require, implementors to treat an Ada package and its sub-packages as a single unit for purposes of analysis. However, standard Java provides no usable chunk size that is both larger than the package and smaller than the entire program. Therefore, I reached into my past experience and built a module system in order to allow the thread coloring analysis to use modules as its analysis chunks.[4]

[3]Note that CVS modules are simply a collection of files, with no access control or other semantics.

[4]The design and implementation of the Fluid module system was greatly improved through discussion with Dr. William Scherlis, Dr. Jonathan Aldrich, Edwin Chan, and Donna Malieri.

3.3 Designing the Fluid module system

3.3.1 Why design yet another module system?

There have been many prior module systems. Prior systems, however, have not addressed the problem with the Fluid group's particular combination of needs.[5] The Fluid module system's combination of old ideas, one new idea, and careful pragmatic design provides a useful and usable capability with significant benefit both for software developers (information hiding, control of potential coupling, composition) and for analysis developers (compositional reasoning across modules, inference within modules).

The Fluid group chose to work with the Java language.[6] Java alone, however, does not solve the problem of information hiding. Current Java mechanisms for information hiding are adequate for projects that fit in a single package. When the project grows beyond a single package, however, types that must be visible to other packages must be declared to be `public`, thus making them visible *everywhere* outside that package[7]. Examples of this issue abound in the Swing API. For example, consider some of the many Swing `public` methods that users shouldn't call—`JComboBox.firePopupMenuWillBecomeVisible()` "This method is `public` but should not be called by anything other than the UI delegate." and `JComboBox.actionPerformed()` "This method is `public` as an implementation side effect. do (sic) not call or override." A larger scoping mechanism such as the Fluid module system permits information hiding on a larger scale.

Java package's limited support for visibility control is not their only deficiency. Our experiments with inference-based analyses (see Section 3.5.2) suggest that packages are not large enough to satisfy program analysis tools' need for large analysis chunks. Further, the interfaces between packages are too large.

Java visibility rules Recall that Java offers only four visibility choices for members: `public`, `protected`, *no modifier*, and `private`. `Public` entities are visible throughout the entire program. `Protected` entities are visible within their own class, to other classes within the enclosing package, and to subclasses in other packages. No modifier entities (sometimes known as "package private" or "default visibility") are visible within their own class and to other classes within the enclosing package, but nowhere else. Finally, `private` entities are visible only within their own class. The lack of a larger-scale information hiding mechanism is the source of comments like the ones referenced above.

3.3.2 Design criteria

The Fluid group's concern with information hiding and coupling control for large scale programming, combined with our need for composable scalable analyses led us to select our specific criteria for the design of the module system. This section presents the important criteria that influenced the design of the Fluid module system. For a comparison of the Fluid module system with other module systems, see Section 3.6. To skip directly to the design itself, go to Section 3.4.

Appropriate notion of "Connectivity"

The Fluid module system uses compile-time visibility as its notion of connectivity. This choice, in combination with explicitly identified interfaces, allows control of potential coupling. By avoiding the need to extend the type-system of the underlying language, we improved the module system's ability to inter-operate with existing tools, and also reduced the effort required to support the module system.

[5]The Fluid group is the research project at Carnegie Mellon University of which the author has been a part.

[6] Our techniques apply equally to other popular imperative languages. A similarly designed system could be built for other languages (such as Ada or C#) by applying the principle that the module system can remove compile-time visibility that the language would otherwise have allowed, but may never increase visibility beyond that defined by the underlying language. This approach allows the module system to enforce limitations on potential coupling with minimal impact on the source language.

[7]One possible explanation for this deficiency is that Java packages appear to be focused more on namespace management than on visibility management.

Visibility control

Module hierarchy style The Fluid module system uses a tree-structured module hierarchy by default. The visibility implications of this hierarchy are analogous to those of lexical nesting in languages like Java or Ada. Strict tree-structure, however, is sometimes too strong a constraint. The Fluid module system supports more complicated nesting, with increased programmer effort when departing from straight-forward trees. Tree structures also degrade gracefully into simple flat structures. After all, a 'tree' consisting solely of leaves *is* a flat structure. The Fluid module system also supports nesting of modules within layers and of layers within modules.[8] This allows an additional level of flexibility when necessary.

It is also possible to decouple the module structure from the structure of the module name-space. This need arises because the programmer may not know the module structure ahead of time when retro-fitting onto large existing programs. The use-case in the section on “Support for incremental adoption” on the next page motivates this criterion more directly.

Explicitly identified interfaces The Fluid module system identifies the interfaces between modules without ambiguity. Non-interface items are hidden from outside modules.

Selective export The Fluid module system allows partial export of visible entities from a class, package, or module. It is often the case that only a subset of a library class's public methods are intended for use by clients. Other public methods may be intended only for internal use within the library and should not be invoked from client code.

Selective import To reduce potential for coupling, the Fluid module system supports expressing policy that restricts the possibility of imports from certain other modules. The designers of a module containing a real-time subsystem, for example, might mandate a policy that no code inside that subsystem import any entities from the GUI module. The Fluid module system enables both expression and checking of such policies.

Support existing code

The Fluid module system accommodates existing code without changes. Specifically, existing code compiles and executes without modification. Placing code within a module may require code changes, however, either to indicate said placement or to adjust for visibility changes. The module analysis may, however, reject code that would have been accepted by the Java compiler.

Stand-off module definition The Fluid module system also accommodates opaque interfaces. It is possible to define a library module and its interface without specifying the internal structure or code of the module. This is necessary to support framework and library development where client code may not yet exist, or where client developers do not have access to library or framework internals.

No language changes Additionally, the Fluid module system works with existing tools. It does not make incompatible changes to existing Java source code, nor does it make non-compiling code compile without errors. We achieve this goal by selectively removing visibility that would be otherwise legal under standard Java semantics; we avoid adding visibility that would not be present under standard Java semantics.

No source code changes Code that is not placed inside a module does not require any change at all. This means that the Fluid module system accommodates unchanged classes as well as those placed within a module.

Scalability criteria

Large modules To support composable scalable static analysis over large chunks of very large programs, the Fluid module system permits both many compilation units per module and many modules per program.

[8]The prototype assurance tool does not, but obviously should, include an implementation of this combination of features.

Support for multiple composable analyses To support composable analyses, we need both analysis cut-points and also boundaries to divide on. Further, the boundaries should both make sense to developers and support developer needs as well as analysis tool needs. The annotations for our various analyses serve as cut-points. The Fluid module system is expressly designed to support analyses that operate one module at a time by providing the necessary boundaries, and to be able to compose the analysis results from multiple modules. Module-at-a-time operation provides a convenient analysis chunk size that is both larger than a single compilation unit and smaller than whole-program analysis. Many analyses, especially those that are inference-based, can provide greater benefit to the end user when they operate on larger chunks of code. Whole program analysis, however, fails to scale well to multi-million SLOC programs. Module-at-a-time analysis provides a useful in-between scope for analysis. This is one of the major drivers of our design. Composability of the per-module results enables scalability to very large programs.

Support for incremental adoption Because the Fluid module system is an extra-linguistic tool—use of which is optional—rather than a language change, it supports introduction of modules without any change to the existing package structure. This is true even for cases where an existing body of code has individual packages that contain classes that should logically be in distinct modules. If the Fluid module system were to require "all-or-nothing" module-ization, it would be unusable in the real world.

The Fluid module system was created partly in response to our team's experience performing program analysis case studies on real-world third-party source code. The programs we have analyzed in the field range from relatively small stand-alone programs (10KSLOC) up through multi-million SLOC server packages from well-known—but unnamed due to non-disclosure agreements—vendors. Many of these programs are large enough that, on consulting with the vendor's development team, we have chosen to analyze only smaller portions in one visit—typically a few hundred KSLOC. This immediately begs the questions: "How shall we carve off a few hundred-thousand lines of code for analysis?" and "How can we be sure that these analysis results are sound, contingent on assumptions about the remainder of the program that will remain as unsatisfied proof obligations?" This experience has convinced us that a successful module system should provide support for incremental adoption with contingently-sound reasoning, all at a reasonable level of effort.

A typical scenario on arrival for an in-field case study might be:

- Discuss the structure of the program to be analysed with its developers.
- Identify a few portions of the program for which analysis results would be of greatest interest.
- Tentatively identify a candidate module structure for those portions of the program.
- Insert the annotations to create those modules and their interfaces.
- Analyze code and produce initial results.

Note that developers are generally unwilling to modify either the architecture or the package structure of their application. The module system must map into the developer's structure, rather than the other way around. Furthermore, any changes larger than adding annotations or comments to the code under analysis are viewed with great suspicion.

Later on in the case-study analysis process, we sometimes discovered that the candidate module structure was either wrong or incomplete. Alternatively, we moved on to analyze additional portions of the program. This process often involved building additional module hierarchy by wrapping modules created earlier with newly created parent modules, adding additional modules, etc. The key observation here is that we are rarely able to elicit a module structure that is either complete or correct at the beginning of an engagement. Therefore we support incremental addition of modules both at the leaves of the hierarchy and near the root of the hierarchy.

This process led us to include annotations whose semantic result is identical to those intended to support module hierarchies defined in advance, but whose syntax and expression eases the process of adding the annotations on the fly. The hierarchy is already present in the code, but the annotations—and the Fluid module system—are late to the party, and must accommodate this fact.

Query support

The Fluid module system provides support for useful queries on the content and interface of modules[9]. These queries might be performed either at runtime via reflection, or ahead of time as part of the operation of a configuration tool, IDE, or other development support tool. Interesting queries might include:

- What is the exported API of this module?
- What does this module import from other modules?
- Given a module-of-interest along with one or more client modules, what items from the module-of-interest are actually used by the clients?
- Given a client module and the APIs it could import in some program, what is the difference between the maximum set of items it *could* import and the set of items it *does* import? What is the difference between the set of items it imports and those it uses?

There are, of course, many other possibilities.

Validation

The Fluid module system provides compile-time validation that the as-written code is consistent with the expressed module structure. When that module structure also reflects (part of) the intended structure of the architectural model of the program, this consistency check provides valuable assurance that the as-written code adheres to the model.

3.4 Design concept, rationale & realization

The above design criteria drove the detailed design of the Fluid module system. This section explains in detail our specific design choices and the capabilities they provide; the rationale behind those choices; and their realization in terms of annotations. The individual features we have implemented are not new; our specific combination of features is. Our sole 'new idea' is the use of module boundaries and interfaces as analysis cut-points. This choice permits analysis tools to analyze individual modules and to compose those analysis results, thus scaling the analyses to very large programs.

We have designed and implemented a hierarchical module system for Java that supports compile-time information hiding for users with explicit control over import and export of classes, methods, and fields. This allows users to reduce coupling, and to statically enforce policies with respect to coupling. The Fluid module system also supports compositional reasoning by program analysis tools including composable module-at-a-time analysis and inference within modules. The Fluid module system is similar to but more flexible than that proposed by the JSR294 expert committee in their Spring 2007 Strawman proposal.

The Fluid module system is implemented and working as a part of the Fluid group's prototype assurance tool. In addition to the module analysis itself, we have built two program analyses that use the Fluid module system to support divide-and-conquer analysis of large programs. These are thread coloring (both for code and for data) and whole-module effects (an alternate version of Greenhouse–Boyland effects analysis that computes the transitive may-read and may-write effects for methods).

The contributions of this work are primarily engineering. These include a pragmatic design for a hierarchical module system built on top of Java, with no need to modify any existing code; support for inference-based analyses within modules (novel in scale, but not concept); the observation that nested modules should not have visibility to all the contents of their containing modules if one-module-at-a-time analysis is to be supported; and the observation that module-at-a-time analysis combined with inference-based algorithms can substantially reduce the programmer effort required to annotate programs for program analysis tools.

Early experiments with an inference-based analysis that operates one-module-at-a-time show that it can reduce the number of user-written annotations by approximately 12x when compared to one that operates over smaller chunks. The prototype assurance tool achieves this reduction by treating the exported interface of each module as both analysis cut-points and as the base of the inference. The inference algorithm then

[9]The prototype assurance tool does not include an implementation of this query capability.

determines consistent annotations for non-exported entities inside the module without requiring user-written annotation.

Additionally, program analysis that proceeds one module at a time can load only the public interface of modules other than that currently under analysis. When used in concert with with soundly combinable analyses, this provides scalability to millions of SLOC or more. Programs at this scale are currently beyond the reach of whole-program analysis.

To understand our design choices, it is useful to remember a few key ideas:

- It's all about visibility:
    - Use of modules may restrict visibility compared to language rules without modules; use of modules never adds visibility. Thus, successful compilation with modules implies successful compilation without. Similarly, compilation errors without modules imply compilation errors with modules – although possibly *different* errors.
    - By controlling visibility, the Fluid module system reduces the potential for undesired coupling.
    - Explicitly specified interfaces allow users to specify and limit the external view of a module.
    - The Fluid module system uses a module hierarchy to assist with default visibility control. Users can escape the hierarchy when necessary.
- It's all about analysis:
    - The Fluid module system assumes that modules are "sealed." The code executed at runtime must be the code the module system analyzed. Dynamic loading is supported either through guaranteeing that the loaded code was presented for static analysis (as part of some module) or via unsatisfied proof obligation. A trusted class-loader can ensure these properties.[10]
        - This property of modules is crucial because it enables module-at-a-time analysis with inference inside modules; specifically, it allows the inference-based analysis to be certain that its results will still hold at runtime.
    - Module interfaces act as analysis cut-points. This enables scalable combinable analysis.
- It's all about scale:
    - The Fluid module system supports incremental module-ization. Users can realize benefit from partial module-ization of a program. There is no "big-bang" transition required.
    - The Fluid module system supports backward compatibility. No changes are required for existing code, but without changes users do not get the benefits of modules.
    - Module-at-a-time analysis with inference inside modules reduces the effort needed for model expression, thus making analysis more usable.

3.4.1 Terminology

We begin this more-detailed look at Fluid's module system by defining some frequently-used terminology. The most-local module that contains a Java entity (from the point-of-view of that entity) is that entity's *home module*. A *top-level module* is a module whose immediate ancestor is the Root module. The *public interface* of a module consists of those portions of its content that are exported from that module. For purposes of the discussion that follows, and without loss of generality, we treat Halloran-style layers as though they are an aggregate module with an identified public interface representing all Java entities that can be referenced from outside the layer (or layers). This allows us to discuss modules without further consideration of layers. An *Applications Programming Interface* (API) contains the public interface of one or more modules. Every entity in an API must be part of the public interface of some module.

[10]The prototype of the Fluid module system does not include this feature.

3.4.2 Modules from thirty-thousand feet

This quick overview provides a minimalist statement of the core design decisions and rules embodied in the Fluid module system. The remainder of Section 3.4 examines these rules in greater detail, discusses their implications, and relates them to the work of others.

- Every module has a public interface, which is a subset of the Java entities in that module. Java entities in the public interface of a module are potentially visible outside that module; all other Java entities in the module are hidden from the outside.
- Visibility within a single module follows ordinary Java visibility rules.
- There is a `default` module, which holds all code not explicitly placed in some other module. The public interface of the `default` module consists of all Java entities in that module; this public interface is visible to all code in all other modules—as restricted by ordinary Java visibility rules.
- There is a hierarchy of modules, which is tree-shaped by default. A compilation unit located in a home module somewhere in the hierarchy can see the Java entities inside its home module, the public interface of that home module, the public interfaces of the home module's siblings, the public interfaces of the home module's transitive ancestors, and the public interfaces of those ancestors' siblings. Because the visibility provided by the default tree-shaped hierarchy is sometimes too constraining, modules can also explicitly permit visibility that goes beyond that provided by default, thus "violating" the tree-shaped hierarchy. Modules may also place additional visibility restrictions on some or all of their own public interface, thus denying visibility to some module or modules that had that visibility by default.
- The Fluid module system uses annotations to place Java code into the module hierarchy, and to identify the public interfaces of modules. The system provides a variety of default behaviors intended to reduce the effort needed to module-ize a program.

3.4.3 Local visibility rule

Any reference from code within a particular home module to another Java entity that is located within the same home module follows ordinary Java visibility rules. The rationale for this rule is to provide minimal interference with ordinary Java visibility. Note, in particular, that this rule provides standard Java semantics for all code that is located in the `default` module and refers to other code also located within the `default` module. Thus, there are no visibility changes for code that has not been placed within any defined module or for programs that have no defined modules.

This rule holds for inner classes as well as for compilation units that contain multiple classes. In all cases, the Fluid module system's visibility rules for Java entities nested inside a compilation unit do not change based on the number or nesting of entities within that unit. Because the only visibility changes introduced by modules are a decrease of visibility, use of modules can never make visible entities that were not visible without the presence of modules.

3.4.4 Hierarchy

The Fluid module system supports a hierarchy of modules. The purpose of this hierarchy is to provide a useful default for control of visibility from one module to another. A common question that arises is "When considering code placed in some module, what other modules are potentially visible from here?" Relationships between modules in the hierarchy such as sibling, ancestor, ancestor's sibling and so on give an easily understood default answer to this question. See the import visibility rule on page 50 for details.

The module hierarchy is tree-structured by default. At the top of the tree is the empty Root; no classes may be placed in the Root. All top-level modules are children of the Root. Directly below the Root module is the `default` module, which may never have any child modules. This module holds all Java code that is not assigned to any other module. Java entities placed in the `default` module are treated as being visible to other modules by default[11]—that is, their visibility follows standard Java visibility rules. This matches the default behavior of unannotated code. Also below the empty Root module are zero or more user-defined

[11]This can be changed by use of the `@NoVis` annotation. See the description of the `@NoVis` annotation in 3.4.5 on page 53.

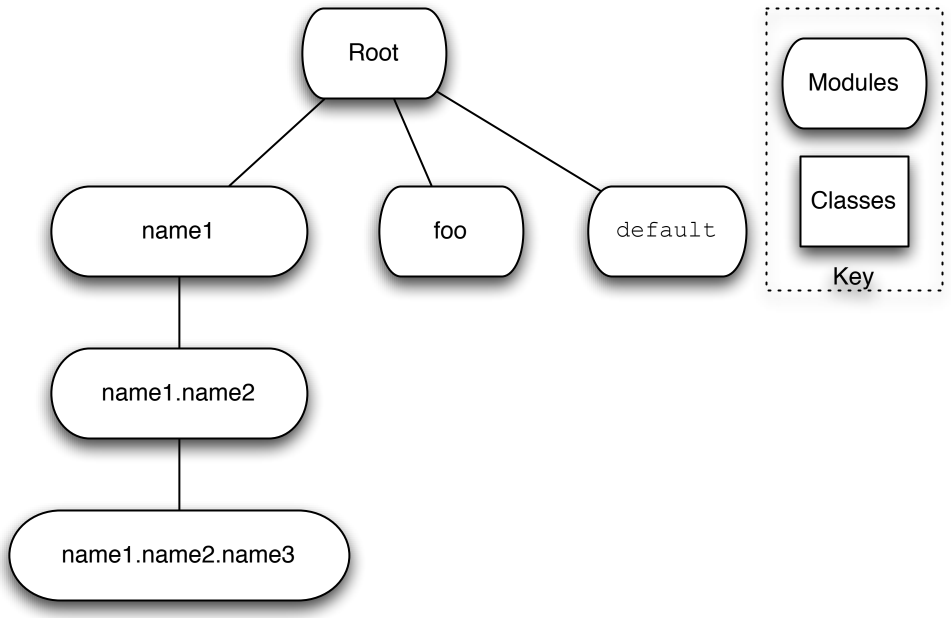


Figure 3.1: Example module hierarchy

modules, each of which may be the head a tree of additional modules. Java entities placed in *non*-`default` modules are *hidden* from other modules, unless specifically annotated otherwise.

These choices for the default visibility of Java entities in the `default` module and for those in *non*-`default` modules are strongly based in the Fluid group's key ideas. Specifically, the choice "visible to other modules by default" for Java entities not placed within a specific module preserves the behavior of ordinary non-module-ized Java code. This backwards compatibility is a key feature of the Fluid module system. The choice "hidden from other modules by default" for Java entities that are placed within a specific module reduces annotation effort by matching well with the most common case; module interfaces are generally much smaller than the number of entities within the module. Most of the contents of most modules remain confined within their home module.

Figure 3.1 shows an example module hierarchy, including the empty Root module, the `default` module, and two other top-level modules both with and without children.

The hierarchy of modules is not an inheritance hierarchy. Instead, it is analogous to the lexical scoping of names in languages like Java and Pascal. Code inside a module has ordinary Java visibility to other code in the same module. In addition, it can see the visible interface of all of its ancestor modules and their siblings. A more detailed presentation of visibility rules and rationale is found in Section 3.4.4 on the facing page.

The Fluid module system only removes visibility from the standard Java model; it never adds visibility. As a consequence, the addition of modules can never make previously illegal code compile (or check successfully in the module analysis). Furthermore, although the Fluid module system is currently implemented as a separate static analysis, we have defined the visibility rules as though they were enforced by the Java compiler.

Note that visibility is neither symmetric, nor transitive. That is, the fact that A can see B does not imply that B can see A (*e.g.* not symmetric). Likewise, the facts that A can see B, and B can see C do not imply that A can see C (*e.g.* not transitive). These properties are necessary to achieve information hiding, and are exactly analogous to the behavior of standard Java visibility.

Running example We use a running example drawn from the AWT and Swing sources (version 1.5) throughout the following sections to motivate the need for the Fluid module system and to illustrate the declaration and usage of its various features. The following running example is presented conversationally.

From package `java.awt.image` we have class `FilteredImageSource`. From package

```
public class FilteredImageSource implements ImageProducer {
    ...
    /**
      * Constructs an ImageProducer object from an existing ImageProducer
      * and a filter object.
      * ...
      */
    public FilteredImageSource(ImageProducer orig, ImageFilter imgf) {...}

    /**
     * ...
     * This method is public as a side effect
     * of this class implementing
     * the <code>ImageProducer</code> interface.
     * It should not be called from user code,
     * and its behavior if called from user code is unspecified.
     */
    public synchronized void addConsumer(ImageConsumer ic) {...}

    /**
     * ...
     * This method is public as a side effect
     * of this class implementing
     * the <code>ImageProducer</code> interface.
     * It should not be called from user code,
     * and its behavior if called from user code is unspecified.
     *
     */
    public synchronized boolean isConsumer(ImageConsumer ic) {...}
    ...
}
```

Figure 3.2: Selected code from class `java.awt.image.FilteredImageSource`

`javax.swing` we have class `JComboBox`. Extracts from their code can be seen in Figure 3.2 and Figure 3.3 on the following page. Note that many methods and fields have been elided for brevity. Pay special attention to the comments on the methods in `FilteredImageSource` and on the `firePopupMenuWillBecomeVisible` method in `JComboBox`. These comments clearly state that they are not intended for use by client code, and should only be invoked from particular places inside the underlying library implementations. We'll use these classes and packages repeatedly to illustrate issues with modules and visibility in large-scale software systems.

We'd like to use the Fluid module system to allow the implementors of the AWT and Swing libraries to invoke the internal methods that are not part of the public APIs of those libraries, while exposing the intended public APIs to the outside world. To accomplish this, we must be able to place classes into modules, define the hierarchy of modules, and indicate which classes, methods, etc. are part of the public interface of those modules.

Default visibility

Figure 3.4 on the next page shows a partial module hierarchy that could be appropriate for AWT and Swing in the context of our running example. The default module visibility rules would permit code in module AWT to see the public interface of module Swing and vice versa. This section discusses the default visibility rules

```
public class JComboBox ... {
    ...
    /**
     * Removes a <code>PopupMenuListener</code>.
     * ...
     */
    public void removePopupMenuListener(PopupMenuListener l) {...}

    /**
     * ...
     * This method is public but should not be called by anything
     * other than the UI delegate.
     * ...
     */
    public void firePopupMenuWillBecomeVisible() {...}
    ...
}
```

Figure 3.3: Selected code from class `javax.swing.JComboBox`

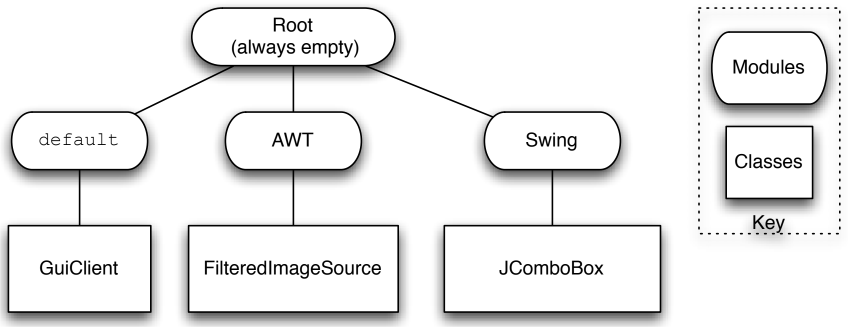


Figure 3.4: Simple Module Hierarchy

that produce this result and the rationale for those rules, describes how our annotations express the module hierarchy, and gives a small example. Later on, in Section 3.4.5 on page 53 we will discuss in greater detail the methods used to distinguish the public interface of a module from those portions that are hidden from the outside.

Export visibility rule A Java entity is visible outside its home module if and only if it is part of the public interface of its home module. Java entities become part of this public interface in either or both of two ways:

- By default, when the Java entity is located in the `default` module.
- When there is an `@Vis` annotation that applies to the entity, making it part of the public interface of its home module.

The entity may also be part of the public interface of some or all of its home module's ancestors, either as a result of an `@Vis` annotation that names one of those ancestors or as the result of one or more `@Export` annotations. This property holds uniformly for type names, method names, fields, and all the rest. See Section 3.4.5 on page 53 for a more complete presentation of the `@Vis` annotation and other methods of controlling visibility of individual Java entities.

This rule is intended to support information hiding by making it possible for modules to expose only a subset of the Java entities they contain. Because Java entities that are not part of the public interface of their home module are hidden outside their home module, code outside that home module cannot make a static reference to those entities. Interfaces and inheritance, however, introduce a few subtleties.

Interfaces In order to implement a Java `Interface` whose definition is imported from another module, the `Interface` itself must be part of the public interface of the remote module. This is an obvious consequence of the export visibility rule. The current module must also have visibility to the remote module (see the import visibility rule on the following page). Less obviously, each individual method you implement must also be part of that public interface (as will be the case by default if the entire interface is marked as being `@Vis`). After all, if the method specifications were hidden, how could the compiler or module analysis tool check to see that they were correctly specified in the implementation? Note, however, that the actual methods need not be visible themselves, nor does the Fluid module system require that the implementing type be part of the public interface of its home module. Thus, it is possible to *dynamically* invoke a method from another module even when the Fluid module system does not allow *static* visibility to it. This is exactly analogous to framework code invoking an interface method whose definition is found in client code. The framework did not have static visibility to the client method—indeed, the client method may not even have been written yet when the framework code was compiled—but the framework may nevertheless invoke the interface method and so execute the client code at runtime.

```
/** @Module A */
/** @Vis */ interface Itf {
    /** @Vis */ public void implementMe();
}

/** @Module B */
class Implementor implements Itf {
    public void implementMe() {...}
}
```

Figure 3.5: Interface example

```
/** @Module C */
...
    Itf dummy = ...something-returning-type-Implementor-as-an-Itf...
...
    dummy.implementMe();
```

Figure 3.6: Interface example extended

Example Returning to our running example, given the definitions in Figure 3.5 (and assuming that module `B` can see the interface of module `A` by the import visibility rule) it should be clear that class `Implementor` has the necessary visibility to implement the `interface Itf`. Suppose that we add a third module `C` to our example (as in Figure 3.6) where `C` can also see the interface of module `A`.

The declaration of `dummy` is OK as far as visibility goes, because `Itf` is part of the visible interface of module A. The call to `dummy.implementMe()` is also OK, because `Itf.implementMe()` is also part of the visible interface of module A. This is true even though the method actually invoked at runtime is `Implementor.implementMe()`, which is *not* part of the visible interface of any module. The rationale for this rule is that modules are used to control static visibility and thus support information hiding. Just as library code may invoke (via dynamic dispatch) a client method that was written long after the library was frozen, so may we dynamically invoke a method to which we have no static visibility.

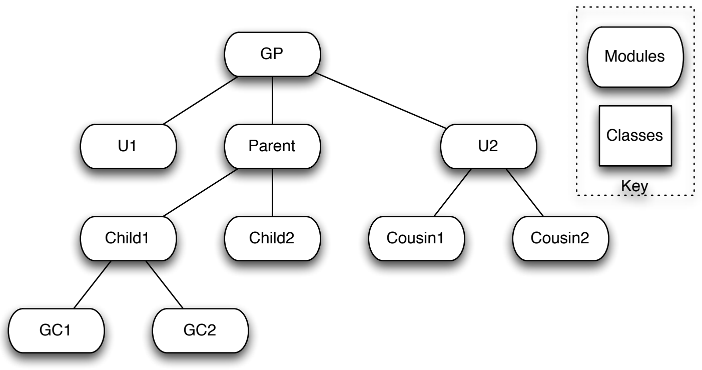


Figure 3.7: Module structure example

Module	GP	U1	Parent	U2	Child1	Child2	Cousin1	Cousin2	GC1	GC2
Child2	X	X	X	X	X	X				
GC1	X	X	X	X	X	X			X	X
GC2	X	X	X	X	X	X			X	X
Cousin1	X	X	X	X			X	X		
Cousin2	X	X	X	X			X	X		
U1	X	X	X	X						

Table 3.1: Module visibility combinations

Inheritance To extend from another type, the type that is the extension must have visibility to the type being extended. This is another natural consequence of the export visibility rule. The child type, however, need not be exported from its home module. As with interfaces, the statically named method must be visible from the call site (either because its declaration is in the same module, or because it is part of the visible interface of a visible module), but the actual member function invoked at runtime need not be visible across modules.

Inner classes The visibility rules apply to inner classes exactly as they do to any other Java entity.

Import visibility rule Any given module can see the visible interfaces of:

- Its siblings in the module hierarchy
- Its ancestor modules' siblings

Because the visible interface of a parent module is at most the union of the visible interfaces of its children, it would be correct (but redundant) to say that each child module can also see the visible interface of its parent. After all, the child module can already see all possible members of its parent's interface as part of its own interface and of the interfaces of its siblings.

Note that the `default` module is automatically a sibling of all of the top-level modules in the entire hierarchy. As a consequence, every visible item in the `default` module (*e.g.* every non-private item in any code not otherwise assigned to a module) is visible to all code in any module.

Previous module systems that use hierarchy [52, 11, 36] (as compared to a flat module space) have supported similar visibility rules for module hierarchies. Our decision to use this visibility rule was based on the principle of least surprise—it matches well with the ordinary behavior of lexical scoping.

Table 3.1 shows the possible combinations of visibility for the module structure shown in Figure 3.7. Combinations marked with an 'X' indicate that the module named in the row has visibility to see the interface of the module named in the column. Thus the 'X' in the upper left-hand corner indicates that module Child2 can see the interface of module GP. Careful use of the `@Vis` annotation can broaden the visibility of items from leaf packages. For example, marking a class declared in module `GC1` with the annotation `@Vis GP` makes that class part of the visible interface of every module on the path from module `GC1` up to and including module `GP`. As a consequence, that class would be visible throughout the entire hierarchy shown in Figure 3.7.

Limitations on code placement

The Fluid module system enforces a limitation on the placement of code—it forbids non-leaf modules that contain code. All non-leaf modules contain only other modules. Allowing code only in leaf-modules simplifies maintaining the guarantee that all modules can be analyzed individually, with access only to the exported interfaces of other modules (but not the code contained therein, if any).

The issue is that the various Fluid inference algorithms must see all the code 'inside' an analysis chunk in one go. Allowing code in non-leaf modules would require either (a) specifying downwards cut-points or (b) analyzing multiple modules (specifically, any non-leaf module that contains code along with all its children) in a single analysis pass. Choice (a) complicates both explanation of the system and expression of analysis cut-points. Choice (b) reduces the system's potential for scalability at analysis time. We chose to forbid non-leaf modules that contain code because both of the alternative choices are unattractive.

This limitation may appear draconian, but is insignificant in practice. The module system could transform a module hierarchy that contains code in non-leaf modules into one that has code only in leaf modules using a simple mechanical transformation:

1. For each non-leaf module M that contains code, create a new leaf module M' whose parent is the original non-leaf module.
2. Move all code from module M into the newly created leaf module M'.
3. Take all the code just moved and make it the visible interface of the newly created leaf module M'.

All visibility is now exactly what it would have been with code in the non-leaf module, but the module system has maintained the property that code is in leaf modules only.[12]

A consequence of this transformation affects annotation of code for analyses that use module boundaries as cut-points. To form an analysis cut-point, code at the module boundary must carry annotations sufficient to express the desired properties at that cut-point. So moving code from a non-leaf module to a leaf module increases the number of annotations to write. On the other hand, the transformation allows analysis to proceed a single module at a time. Without this transformation, any analysis that performs inference within analysis chunks would have to process the non-leaf module containing code along with all its descendant modules in a single analysis pass. We chose to trade a modest number of additional annotations for a smaller granularity of analysis.

Module placement annotations

Because the Fluid group's typical use-case involves layering modules on top of preexisting programs, the Fluid module system supports two styles of module naming. All annotations that permit a module name accept the same syntax. Module names can be either simple names or dotted names; both follow the usual Java syntax rules.

@Module <name> This annotation declares the existence of the named module, and maps the annotated compilation unit into the named module. If the named module has a dotted name (*e.g.* `name1.name2.name3`), the annotation declares the existence of a module hierarchy (as in Figure 3.8). Note that the names of the three modules so declared are "`name1`," "`name1.name2`," and "`name1.name2.name3`." This is exactly analogous to the naming of Java packages.

Experience to date suggests that the dotted-hierarchy style of module naming is most useful for hierarchies that are designed before the addition of modules to the source code. When adding modules to a preexisting program (especially to a program not yet understood), it is generally preferable to use simple names for leaf modules, and to then build the module hierarchy using the module-wrapping annotations. Because the dotted style of naming encodes a specific hierarchy, it requires committing

[12]The prototype assurance tool lacks an implementation of this transformation.

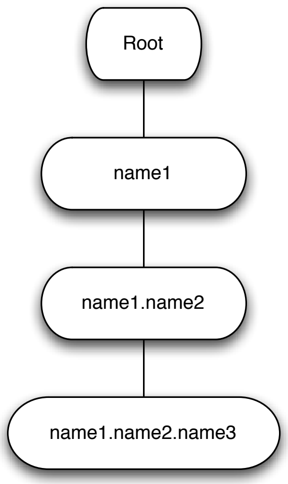


Figure 3.8: Hierarchy from `@Module name1.name2.name3`

to that hierarchy before placing the annotations in the code. Conversely, naming modules with simple names and using the wrapping annotations to build the hierarchy somewhat delays the commitment.

@Module <name1> contains <name2> This annotation is intended for wrapping preexisting modules with newly created ones. It declares the existence of module `name1` and also indicates that module `name1` is the parent of module `name2`. It is permitted only in `package-info.java` files, or their equivalent stand-off annotation files. Note that this annotation *does not* declare `name2`; its declaration must be provided by some other annotation. The effect of the annotation (in terms of the module hierarchy) is no different than using dotted names to create hierarchy; it simply supports adding the parent module later, without having to edit preexisting module annotations in any way. Because the module hierarchy is a tree, annotations that establish multiple parents for any other module are in error, as are annotations that produce a wrapping loop.

Example Returning briefly to the running example, Figure 3.4 on page 48 shows a simple module hierarchy that would be suitable for AWT and Swing[13]. It shows our two example classes located within the appropriate modules, but is silent about what constitutes the visible interface of each module. To place our two example classes into modules as shown in the figure, we must add suitable `@Module` annotations to each of them, as follows:

```
/** @Module AWT */
public class FilteredImageSource implements ImageProducer {
   ...
```

and

```
/**@Module Swing */
public class JComboBox ... {
    ...
```

Note, however, that we have not yet specified anything about the public interface of the modules.

[13]Of course, the AWT and Swing modules would also contain the many other classes that make up these frameworks. Those other classes are elided for clarity of presentation.

Rationale for hierarchy

Consider first the case of a module hierarchy consisting of the (empty) Root module and its immediate children, with no other modules. With the three visibility rules given above, this is equivalent to a flat module structure. It permits each module to hide its implementation while exporting selected Java entities to the outside world. This is a useful extension to standard Java visibility using packages, but provides only one additional level of hiding.

Experience with large programs suggests that they are often constructed by combining subsystems, each of which consists of a group of sub-subsystems, and so on. To allow the smaller chunks (the sub-subsystems) to hide their implementations from each other while sharing interfaces that they do not export to still higher levels, the Fluid module system needs a more general mechanism. A tree-structure is a natural fit for this case.

The Fluid group's experiments with Electric (see Section 3.5.2) and our experience examining programs in the field suggest that the module structure for programs of moderate size—150KSLOC or so—are likely to be fairly flat. Larger programs are the likely users of deeper module hierarchies.

3.4.5 In/Out visibility control

The Fluid module system supports a variety of mechanisms for controlling both import and export visibility. First, the requirement for explicit export of all APIs allows designers to hide non-exported code from other modules; any Java entity that is not explicitly exported is hidden outside its home module. Secondly, the tree-structured hierarchy provides default visibility rules that limit coupling between widely separated modules except through highly constrained interfaces.

Individual modules can further limit both incoming and out-going visibility, even when the normal tree-structure would permit it. Developers of modules may refuse to export portions of their module's API to specifically named modules that would ordinarily have visibility to that API. For example, the annotation "`@NoExport` some-named-things `to` list-of-modules-I-don't-trust" would deny access to some-named-things from all of the listed modules, even if those modules would ordinarily have visibility to the named things. On the incoming side, a developer may choose not to permit access to otherwise visible items. For example, a module's maintainers might choose to enforce the rule that "nothing within this module may reference anything from the Realtime module," or "hide all external items named george."

Controlling export visibility

@Vis <name>? This annotation marks the annotated Java entity as being part of the visible interface of some module.[14] If the name is absent, it will be the Java entity's home module. If the name is present, the entity is part of the visible interface of each module from the annotated entity's home module, up to and including the named module. The annotation is incorrect if the named module is not an ancestor of the module containing the annotation.

If, for example, the annotated entity is located within the module named `edu.cmu.cs.fluid.example`, the annotation `@Vis edu.cmu.cs` would indicate that the entity is part of the visible interface of the modules `edu.cmu.cs.fluid.example`, `edu.cmu.cs.fluid`, and `edu.cmu.cs`.

@NoVis This annotation declares that the annotated Java entity is *not* part of the visible interface of its home module – or in fact, of *any* module. Typically used to over-ride class level `@Vis` annotations for members that should not be visible, or to restrict visibility of a Java entity located in the `default` module so that it is visible only to other Java entities located in the `default` module.

For each of the above annotations, when `@Vis` (or `@NoVis`) is applied to a class (or interface, or enum), the declared visibility also applies to all public members of that class. This default visibility can be over-ridden with explicit `@Vis` or `@NoVis` annotations located on the individual members so annotated.

@Export <names> from modName This annotation declares that the named Java entities are part of the visible interface of the named module. For such an annotation to be legal, each named entity must

[14]The "?" indicates that the bracketed item may be absent.

already be part of the visible interface of some immediate child of the named module. Two important consequences of this restriction are that (a) `@Export` does not permit expanding the visible interface of a child module beyond what the child module intended to export, and (b) `@Export` can only promote visibility of a Java entity by a single level in the module hierarchy. The Fluid group chose these restrictions in order to prevent the use of this annotation to "reach inside" someone else's abstraction. Visibility is an important commitment for the designer of a library, framework, or other interface. Later programmers should not be permitted to expand the visible interface of a module without the concurrence of the designer (or at least, without realizing they are "breaking the rules").

This annotation is intended to support after-the-fact wrapping of modules and export of Java entities; it may appear only in `package-info.java` and equivalent `.xml` files. The semantic effect of this annotation is identical to `@Vis`. The only difference is that `@Export` plays well with the "`@Module ... contains ...`" annotation when defining the interface of a newly added module that wraps preexisting modules.

@NoExport {of <names>}? from definingModName to <deniedModNames> This annotation denies export of the named Java entities to the named modules. When the "`of <names>`" clause is present, each named Java entity must be part of the visible interface of the defining module. When the "`of <names>`" clause is not present, the `@NoExport` applies to the entire visible interface of the defining module. The list of denied modules is permitted to name modules that are not known to the system. It is an error for the `definingModName` to be a member of the list of `deniedModNames`.

This annotation allows the designer of a module to control the visibility of his module's interface more finely than the default visibility rules allow. For example, this annotation allows a designer to forbid a sibling module from using some or all of his module's interface, even though the default rules would otherwise allow the sibling module to see the denied items.

Typical use of this annotation is to reduce potential coupling by restricting the potential visibility of portions of the defining module's interface.

Example Returning briefly to the running example (see Figures 3.2 and 3.3), we must add suitable `@Vis` and `@NoVis` annotations to the classes and methods shown in order to fully support the desired visibility of our example code. The new version of the code is seen in Figure 3.9 on the facing page and Figure 3.10 on page 56.

As a result of the `@Vis` annotation on line 2 of Figure 3.9 on the facing page and the redundant[15]—but legal—`@Vis` annotation on line 10, clients of the AWT library can refer to class `FilteredImageSource` , and can invoke its constructor. As a consequence of the `@NoVis` annotations on lines 22 and 34, the module analysis will report an error on any attempt to call `addConsumer` or `isConsumer` from client code, as it is not exported from the `AWT` module and so is hidden from client modules. Likewise, class `JComboBox` and its method `removePopupMenuListener` are part of the exported interface of module `Swing` (as a result of the `@Vis` annotation on line 3 of Figure 3.10 on page 56 and the redundant but legal `@Vis` annotation on line 9), and so may be referred to and invoked by client code. The `@NoVis` annotation on line 17 indicates that `JComboBox.firePopupMenuWillBecomeVisible`, however, is not exported so clients that try to invoke it will get an error from the module analysis. This visibility and these errors statically enforce the intended visibility of these classes and methods as documented in the original author's comments in the JavaDoc. Standard Java does not provide any mechanism to state or enforce this constraint.

Controlling import visibility

It may be desirable as a matter of policy to reduce coupling by restricting what other modules may be referenced from a particular module (or package, or class, etc.). To this end, the Fluid module system supports the following annotation:

@ForbidImport {of <names>}? from <moduleNames> This annotation declares that the annotated scope has restricted importation from the named modules. If the "`of names`" clause is present, the named

[15]This annotation is redundant because the class-level `@Vis AWT` already marks all public entities in the class as being visible.

```
/** @Module AWT
 *  @Vis AWT
 */
public class FilteredImageSource implements ImageProducer {
    ...
   /**
     * Constructs an ImageProducer object from an existing ImageProducer
     * and a filter object.
     ...
     * @Vis AWT // annotation is redundant, but legal
     */
    public FilteredImageSource(ImageProducer orig, ImageFilter imgf) {...}

    /**
     * ...
     * <p>
     * This method is public as a side effect
     * of this class implementing
     * the <code>ImageProducer</code> interface.
     * It should not be called from user code,
     * and its behavior if called from user code is unspecified.
     * @NoVis // Not exported from module AWT!
     */
    public synchronized void addConsumer(ImageConsumer ic) {...}

    /**
     * ...
     * <p>
     * This method is public as a side effect
     * of this class implementing
     * the <code>ImageProducer</code> interface.
     * It should not be called from user code,
     * and its behavior if called from user code is unspecified.
     * @NoVis // Not exported from module AWT!
     */
    public synchronized boolean isConsumer(ImageConsumer ic) {...}
     ...
    }
```

Figure 3.9: `FilteredImageSource` code with visibility annotations

```
/**
 * @Module Swing
 * @Vis Swing */
 public class JComboBox ... {
     ...
     /**
      * Removes a <code>PopupMenuListener</code>.
      * ...
      * @Vis Swing // annotation redundant, but legal
      */
     public void removePopupMenuListener(PopupMenuListener l) {...}

     /**
      * ...
      * This method is public but should not be called by anything
      * other than the UI delegate.
      * @NoVis // Not exported from module Swing!
      */
     public void firePopupMenuWillBecomeVisible() {...}
     ...
}
```

Figure 3.10: `JComboBox` code with visibility annotations

Java entities may not be imported. If no "`of names`" clause is present all importation from the named modules is forbidden. This annotation removes visibility from items that would otherwise be considered visible through the default visibility rules.

This feature is typically used to enforce designed restrictions on coupling, such as "no one within the Realtime subsystem may import anything from AWT or Swing."

3.4.6 Non-tree-shaped module hierarchy

Although tree-shaped hierarchies are a common and convenient structuring approach for large programs, there are times when a tree is too restrictive. Consider, for example, the case of two small subsystems that need to collaborate in order to provide some desired functionality, without exporting their interfaces to the world at large. If these subsystems are not located within the same sub-tree of a program, they will not have visibility to each other unless they expose the necessary APIs to all top-level modules.

For example, consider modules `Child1` and `Cousin2` in Figure 3.7 on page 50. In a strictly tree-structured hierarchy, these modules can only see APIs exported from `Parent` (for items in `Child1`) and `U2` (for items in `Cousin2`). But such APIs are quite high in the hierarchy, and are visible to modules throughout the entire program. One could argue that the need for such collaboration between widely separated modules indicates that it is time to refactor the program in order to place non-public collaborators close to each other in the visibility tree. This would, however, violate the Fluid group's requirement that the module system successfully handle real code without major refactoring.

For this reason, the Fluid module system supports exceptions to strict tree-structured visibility. A module can explicitly export some or all of its API to a specifically named module or modules that would not otherwise have visibility to that API. Using this facility, `Child1` can export parts of its interface directly to `U2`, and *vice versa*. We chose to require explicit export for these non-tree structured cases in order to avoid unanticipated or undocumented coupling between widely separated modules. In cases where large numbers of non-tree-structured exports are required, it may be desirable to consider using a layered model[16] rather than a module-based hierarchy.

[16]See Halloran's layers concept.

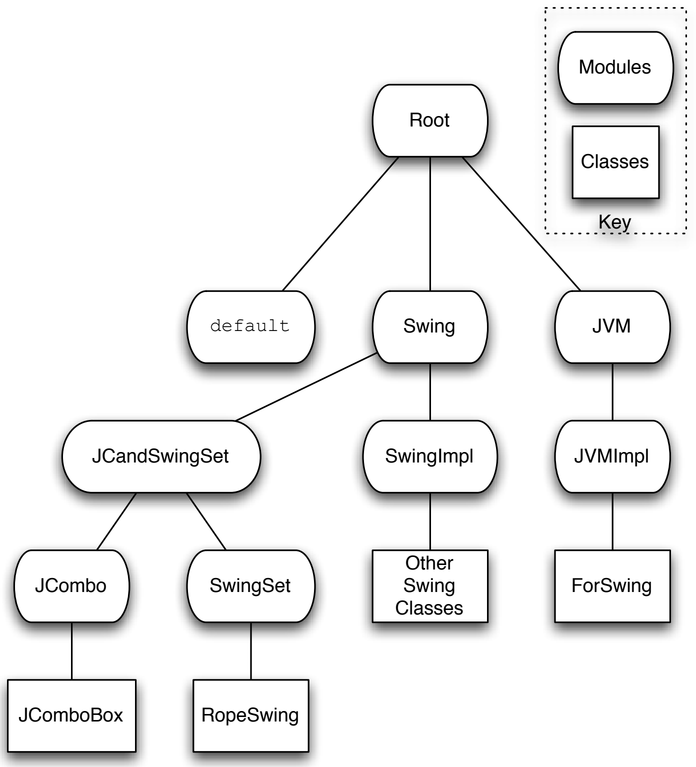


Figure 3.11: Hypothetical JVM and Swing implementation module hierarchy

To support this kind of non-tree-structured dependency, the Fluid module system provides the following annotation:

@Export <Java_Names>? to <Module_names> This annotation indicates that the named modules are granted visibility to the named Java entities. If the `Java_Names` clause is omitted, the named modules are granted visibility to the entire public interface of the annotated module. There may be many such annotations present in a single module. It is an error if the Java entities named in the `Java_Names` clause are not part of the visible interface of the module that contains the annotation. This annotation may be placed with the annotations for the module,[17] in a `package-info.java` file (if all classes in the package are part of the same module), or in an ordinary Java file.

Example

We now expand our imaginary Swing example to illustrate our method for handling these cases. Suppose that an unrelated portion of the Java Library—perhaps the JVM itself—exports some methods that the Swing developers wish to use. After coordination with the developers of the other subsystem to ensure that the dependency will not adversely impact program builds, will not introduce undesired coupling, and that the other development team is willing to commit to the interface in question, the various developers modify the module annotations to give the Swing implementation visibility to the appropriate classes and methods.

Figure 3.11 shows our hypothetical module hierarchy. The Swing implementation code simply adds an ordinary Java import statement to their code. The JVM team adds annotations to their class `ForSwing` as follows:

```
/**
```

[17]Where-ever those wind up—this is not yet clear in the Fluid module system implementation as of May 2008.

```
2  * @Module JVMImpl
3  * @Vis JVMImpl
4  * @Export ForSwing, importantMethodForSwing() to SwingImpl
5 */
6 class ForSwing {
7   public void importantMethodForSwing() {...}
8 }
```

Note that class `ForSwing` is part of the visible interface of the JVM implementation, but is hidden from arbitrary client code. Further, the JVM implementors have placed the `@Export` annotation directly on the class being exported on line 4 in the listing above, rather than in a more remote file.

Rationale

The Fluid module system requires the exporter of an interface to agree that they will make the export, rather than allowing importers to grab hold of a class or method without the exporter's permission. Stepping outside the always-present tree-structured visibility represents an interface commitment to a group that the developers of a subsystem do not ordinarily support. Accordingly, the Fluid module system supports doing this when necessary, but only with the agreement of the developers who will be required to support the interface in the future and who will be responsible for managing the dependency.

3.4.7 Support for existing code

The `default` module contains all Java entities that are not declared to be part of some other module. Unless otherwise modified, all Java entities in the `default` module are part of the visible interface of the `default` module. The `default` module is a top-level child of the Root of the module hierarchy. Since all Java entities not placed in other modules are placed in the `default` module, all unannotated Java code winds up in the `default` module. As a result, all completely unannotated Java code has ordinary visibility to all of its own parts. The Fluid module system is thus fully backwards-compatible with ordinary Java when programs are entirely unannotated.

When a Java system is partially placed into modules, the unannotated portions of the program are placed automatically into the `default` module. As a result of the import and export visibility rules, Java entities in the `default` module are potentially visible to code in any module anywhere in the program, subject only to the restrictions of ordinary Java visibility.

Code located in the `default` module has visibility to the public interfaces of all top-level modules and, of course, to all of the code whose home module is the `default` module. This restricted visibility allows implementors of libraries, for example, to make the API of their library visible to unannotated code while hiding the implementation from the outside world. Further, this hiding can be accomplished without requiring any annotation in client code.

Example: a library API

To illustrate the use of modules to expose a library's API without exposing its implementation, we return to our running example. Let us suppose that the unannotated class `GuiClient` wishes to invoke some methods from the AWT and Swing frameworks; say **`new`** `FilteredImageSource(...)`, `removePopupMenuListener(...)` and `firePopupMenuWillBecomeVisible()` for example. Figure 3.12 shows the module hierarchy. We assume the `@Module` and `@Vis` annotations from Figure 3.9 on page 55 and Figure 3.10 on page 56 are present. Sample code found in `GuiClient` might be:

```
class GuiClient {
  ...
  ImageSource exIS = new FilteredImageSource(...);
  exIS.addConsumer(...);
  someComboBox.removePopupMenuListener(...);
  someComboBox.firePopupMenuWillBecomeVisible();
```

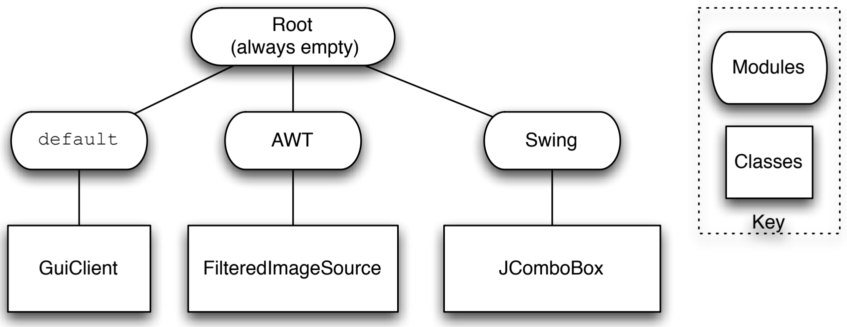


Figure 3.12: GUI usage example

```
7    ...
8  }
```

Since `GuiClient` is located in the `default` module, it can see the visible interfaces of modules `AWT` and `Swing` because they are siblings of the `default` module. The invocations at lines 3 and 5 are OK, because **`new`** `FilteredImageSource(...)` is part of the visible interface of module `AWT`, and `removePopupMenuListener()` is part of the visible interface of module `Swing`. Conversely, the invocations at lines 4 and 6 are in error, as `addConsumer()` is *not* part of the interface of `AWT`, and `firePopupMenuWillBecomeVisible()` is *not* part of the interface of `Swing`.

Example: using modules to hide interfaces between subsystems

Suppose we have two subsystems each of which implements a portion of the Swing GUI library – `JCombo` and `SwingSet`.[18] Suppose further that these subsystems need to communicate via an interface that they do not wish to share with the rest of the Swing library (much less the outside world). For this example, assume that `SwingSet` contains the class `RopeSwing` seen here:

```
public class RopeSwing {
   ...
   /** Intended for use only by JComboBox (don't ask why!) */
   RopeSwing(int i) {...}
   ...
   /** Intended for use by the general implementation of Swing,
    *  but not by client code */
   void swingOnTheRope() {...}
}
```

In addition, let us suppose that the visible methods of `JComboBox` are the entire public interface of module `JCombo`, and that the constructor `RopeSwing(`**`int`**`)` is to be used only in `JComboBox`. Method `swingOnTheRope`, however, is allowed to be invoked from any part of the Swing implementation, but not by user code. To enforce the desired visibility, we need to build a module hierarchy like the one seen in Figure 3.11 on page 57. By changing the visibility annotations in `JComboBox` and `RopeSwing` as shown in Figure 3.13 on the following page, we achieve the desired visibility.

Note that although `JComboBox` is now placed in a module nested well below the Root, most of its methods are exported as part of the interface of the top-level module `Swing`, thus making them visible to client code. The method `firePopupMenuWillBecomeVisible()` is now exported as part of the interface of module `JCandSwingSet`, making it visible to the rest of the Swing implementation, but not to client code.

[18]Both the names and the example are purely hypothetical. To the best of my knowledge, these subsystems do not exist.

```
/**
 * @Module SwingSet
 * @Vis JCandSwingSet // Class visible to siblings of JCandSwingSet
 *                    // (e.g. all of Swing Implementation)
 */
public class RopeSwing {
   ...
   /** Intended for use only by JComboBox (don't ask why!)
    * @Vis SwingSet */
   RopeSwing(int i) {...}
   ...
   /** Intended for use by the general implementation of Swing,
    *  but not by client code.
    *  @Vis JCandSwingSet  // redundant with annotation on class */
   void swingOnTheRope() {...}
}

/**
 * @Module JCombo
 * @Vis Swing */
 public class JComboBox ... {
    ...
    /**
     * Removes a <code>PopupMenuListener</code>.
     * ...
     * @Vis Swing // annotation NOT redundant
     */
    public void removePopupMenuListener(PopupMenuListener l) {...}

    /**
     * ...
     * This method is public but should not be called by anything other
         than
     * the UI delegate.
     * @Vis JCandSwingSet // Exported from JCandSwingSet, so visible to
     *                    // swing implementation, but not
     *                    // exported from module Swing!
     */
    public void firePopupMenuWillBecomeVisible() {...}
    ...
}
```

Figure 3.13: `SwingSet` example code

3.5 Case study and validation

This section discusses benefits the Fluid module system provides to the prototype assurance tool. We first discuss two analyses that take advantage of modules to provide an improved user experience, both in terms of scalability to larger programs and in terms of reduction of the number of annotations a user must write. This is followed by a detailed discussion of the costs and benefits of using the Fluid module system in concert with thread coloring during a case study on a medium-sized—140KSLOC—program.

3.5.1 Analyses that take advantage of the Fluid module system

Scalability through partial loading of source code

Partial loading of source code has benefits when analyzing large programs. The Fluid group's current Eclipse-hosted analysis tools fill the memory of a 32-bit JVM when analyzing programs between 100KSLOC and 200KSLOC[19]—but there are interesting programs that are much larger than this. Multi-million-SLOC programs may well exceed the available resources on realistic hardware even with 64-bit JVMs. Clearly some sort of divide-and-conquer approach to analysis is desirable. Additionally, even when a developer has the resources available to analyze an entire program, a subset of which is of interest, the developer may not wish to wait while the prototype assurance tool analyzes code that is not of immediate concern.

To support scalability, the prototype assurance tool uses modules as its basis for dividing programs, performing partial analysis, and re-combining results. The approach is to load one module in full, load only the specifications of the remaining modules, and analyze. This can be repeated until the entire program has been analyzed. The visible interface of each module is treated as an analysis cut-point. That is, each class, method, or field that is part of the visible interface of a module must carry sufficient annotations to allow analysis of any client module to proceed independently. When analyzing a module, annotations about the code in other modules are treated as unsatisfied proof obligations. Those annotations are checked when analyzing the module that contains them. Thus, analysis of the complete program can proceed one module at a time.

To support composable partial analyses, the Fluid analysis tools are capable of loading code from any module in several different modes:

Whether to load

Required Modules noted as "Required" have all classes loaded according to the Source/Spec selection for that module.

As Needed Classes in modules noted as "As Needed" are loaded only when there is a static reference from some loaded class to a Java entity in that class. If loaded, they are loaded either "As Source" or "As Spec" according to the setting for their module.

How to load

As Source Classes loaded from modules noted as "As Source" have their entire abstract syntax tree (AST) loaded for analysis.

As Spec Classes loaded from modules noted as "As Spec" are partially loaded—the AST for the specification of public methods and classes, public fields, etc. Portions of the AST that would not be visible to another class are not loaded.

To analyze a particular module, the developer doing the analysis loads that module “As Source” and “Require”s the presence of all of its classes; the rest of the program is loaded “As Needed” and “As Spec.” Indeed, these are the commonly used combinations of the two controls.

Module-at-a-time analyses

This module-at-a-time analysis approach allows an analysis tool to rely on inference-based analysis techniques within each module. Inference-based analysis is not a new idea, of course, but its application to composable analysis of large programs appears to be new. Recall from Section 3.3.2 that our annotations serve

[19]As of late fall 2007. Later engineering improvements support analysis of larger programs.

as analysis cut-points. By using the annotations at module boundaries as the fixed base of its inference, the analysis can allow its users to forgo annotation of methods that are not exported from that module. We have implemented two analyses that proceed in this fashion: whole-module effects analysis (an alternate version of Greenhouse–Boyland effects analysis that computes the transitive may-read and may-write effects for methods in a flow-insensitive manner), and thread coloring analysis. This inference-within-modules approach provides a triple benefit: it reduces the number of annotations the user must write, it provides an analysis chunk size larger than per-class or per-package but smaller than whole-program, and it improves analysis quality because there are more program entities for which the analysis can have perfect usage knowledge—these latter are the entities whose uses are entirely confined in their home module.

Enabling assumptions Module-at-a-time analyses depend on three enabling assumptions:

1. The system has closed modules (sometimes called "sealed" modules), where dynamic loading is either not allowed, or is guaranteed to load only classes presented for analysis as "part of" the module. This property guarantees that the tool has analyzed the entire contents of a module, enabling analyses that take advantage of having seen all calls and/or all references to particular fields or methods. The sealed modules property also supports improved semantic guarantees on some analyses (for example the Greenhouse lock analyses, whole-module effects analysis, and data coloring).
2. The loaded classes actually match the source code that was analyzed. This is an obvious requirement for soundness.
3. Annotations on module-boundaries are true analysis cut-points; that is, they are in fact "good enough" to stand as full proxies for the unseen code on the other side of the module boundary.

Note that assumptions 1 and 2 can be checked via a trusted class-loader (not built yet, as it is an engineering effort rather than research). Assumption 3 can be checked by using the module boundary annotations as cut-points; and then checking later—perhaps at class-load time—to ensure that the annotations used were "compatible."[20] If the annotations are both compatible or identical, and are supported by analysis from at least one side, they are certainly true cut-points. When the cut-point describes code not available for analysis—for example commercial libraries, native libraries, or code that simply has not been analyzed yet—assumption 3 on compatibility remains as an unsatisfied proof obligation.

3.5.2 Electric 8.01 case study

To assess the benefit of using module boundaries as analysis cut-points and to assess the cost of dividing a medium-sized program into modules, we applied these techniques to a well-known open-source VLSI-design tool. Electric—which was originally written in C, then translated into C++, then Java—is nearly twenty years old. It is used by a wide variety of design teams and is actively maintained. Version 8.0.1 of Electric contains roughly 140KSLOC, spread over forty-four Java packages.[21] A review of the program's structure shows that it splits quite naturally into four modules:

UI	The User Interface components
Tool	The analysis code of the tool itself (the "`Jobs`") and the various analyses that it performs (*e.g.* design rule checking, logic simulation, feature sizing, *etc.*)
Database	The classes used to represent all data about the circuit the user is building.
Other	A few other packages. This includes the main program in `main.java`, a little bit of initialization code, and a few other pieces that didn't fit in the other three modules.

Counts of annotations

Analyzing Electric requires adding annotations to the program in order to split it into modules and to define the cut-points on which thread coloring analysis will depend. One question we had going in to the study was to ask how many such annotations are needed? We apply three main techniques to reduce the number of

[20]Note that the easiest compatibility check is identity!

[21]A more complete discussion of this case study is located in Section 4.5 on page 90. That presentation focuses on the process and details of the thread coloring analysis. The presentation here focuses on the use of modules.

```
1  /**
2   * @Promise "@Module Tool" for com.sun.electric.tool:*:
3   *
4   * @Promise "@Vis DB" for kill*(**) | modify*(**) | erase*(**) | write
        *(**)
5   * @Promise "@Vis DB" for get*(**) | new(**) | examine*(**)
6   *
7   * @ColorImport com.sun.electric
8   * @Promise "@Color DBChanger" for kill*(**) | modify*(**) | erase*(**)
        | write*(**) | new(**)
9   * @Promise "@Color (DBChanger | DBExaminer)" for get*(**) | examine
        *(**)
10  */
11 package com.sun.electric.tool;
```

Figure 3.14: Example annotations from `com.sun.electric.tool.package-info.java`

annotations required to analyze Electric: annotation inheritance, annotation inference, and scoped promises. These techniques are not new, but they combine to produce a surprising result.

As with the running examples, we now return to a conversational presentation style.

Number of annotations for modules To split Electric into four modules, we must place each compilation unit into a module. We could do this by adding an individual `@Module` annotation to each type. This would mean writing one annotation for each of the several hundred types in the application. Because end-user programmers are unlikely to go to that much effort, we instead use Halloran-style scoped promises (see Section 2.3.10 on page 28 for more information) to accomplish the same effect with fewer annotations. By placing a single annotation in the `package-info.java` file for each package (as seen in Figure 3.14, line 2) we effectively apply the `@Module` annotation to every type in the package. With forty-four packages, this requires forty-four annotations. This could be reduced to four annotations if we supported scoped promises with wild-card selection over packages, rather than only over type names as at present.

Now that Electric is split into modules, we must identify the public interface of each module. Since we are not the original developers of this code, we assume that any Java entity that is referenced across module boundaries in the as-written code is properly part of the interface of its home module. Once again, rather than writing individual `@Vis` annotations for each type, method, and field, we use scoped promises to accomplish the same goal. This could, in theory, be accomplished with a single scoped promise per package, for a total of another forty-four annotations. Sadly, those annotations would each be unreadably long, so we divide them into six annotations per package yielding a total of two-hundred sixty-four additional annotations. Examples of these annotations can be seen in Figure 3.14, lines 4 & 5.

It is important to note that a straight-forward extension to the Fluid group's existing tooling could have created equivalent `@Vis` annotations with only a few mouse-clicks on the user's part. Given a code-base that is partitioned into modules, the Fluid module system can identify every Java entity that is referenced across module boundaries by the as-written code. Indeed, it *must* do so in order to check for references that are not allowed by the visibility rules. Armed with this information, a visibility annotation wizard could use the Eclipse refactoring support to insert the necessary `@Vis` annotations automatically.

Table 3.2 on the next page summarizes the numbers of annotations needed to module-ize Electric. The "Current Number" column shows the actual number of annotations we used to divide Electric into four modules, and to specify the public interface of each. The "Achievable Number" column shows the number of annotations we would use if both of the above improvements were implemented. The current effort required to split Electric into four modules is very small—only 2.2 annotations per thousand lines of source code. With the described improvements, this could be as low as one annotation per *ten thousand* lines of source code. Even without those improvements, it seems reasonable to argue that we have achieved the Fluid group's goal of ease of expression.

Annotation	Current Number	Achievable Number
`@Module`	44	4
`@Vis`	264	10
Total	308 (2.2 Annos/KSLOC)	14 (0.1 Annos/KSLOC)

Table 3.2: Number of module annotations in Electric

Annotation	Current Number	Achievable Number
`@Color`	528	12
other annos	52	9
Total	580 (4.1/KSLOC)	21 (0.2/KSLOC)

Table 3.3: Number of thread coloring annotations in Electric

Number of annotations for thread coloring Each method or field that is part of the public interface of a module—and could thus be referenced across module boundaries—must be annotated with thread coloring information in order to form the desired analysis cut-point. We accomplish this by using a group of scoped promises, as seen in Figure 3.14 on the preceding page, lines 8 & 9. As with the `@Vis` annotations above we use six such annotations for each package, yielding a total of two hundred sixty-four additional annotations. We also need one `ColorImport` annotation per package, along with some additional color declarations[22]. These add another fifty-two annotations. As with the annotations for module-ization, these numbers could be reduced substantially via improvements in support for wild-card matching across packages. Table 3.3 summarizes the counts of thread coloring annotations required for analysis of Electric.

Combining the numbers for both module-ization and thread coloring, we see that breaking Electric into four modules and thread coloring its code required a total of eight hundred eighty-eight (888) annotations. That is 6.3 annotations per thousand lines of source code. If, however, the Fluid group upgraded the prototype assurance tool as outlined above, this could be reduced to a total of thirty-five annotations; 0.25 per thousand lines of source code.

Other module choices Although Electric breaks down nicely into four modules, it is instructive to consider alternative approaches for divide-and-conquer analysis. There are 12,802 Java entities not declared as `private` in the Electric source code. If the module analysis used an inference algorithm that applied only within compilation units, and so only to `private` entities, we would have to annotate every `public` entity in order to complete our analysis. Since it was impractical to do this, even with scoped promises, we now switch from counting *annotations* to counting Java entities potentially referenced across module boundaries. For each module size chosen, we present the number of Java entities requiring annotation in order that analysis may proceed. Our goal is to demonstrate the effect that using module boundaries as analysis cut-points

[22]See the Electric case study in Section 4.5 on page 90 for details

	Annotations for Modules	Annotations for Thread Coloring	Total Annotations	Annotations per KSLOC
All non-`private` entities	0	12802	12802	~86
With Inheritance, Inference & Scoped promises	308	580	888	~6.3
As above, but Scoped promises at Module scope	14	21	35	~0.25

Table 3.4: Combined annotation reduction in Electric

	One Module per CU	One Module per Package	Four 'Random' Modules	Four 'Good' Modules
Non-`private` Entities	12802	12802	12802	12802
Referenced across Module Boundaries	12802	7203	4966	2522
% that are Interface	100%	56.3%	38.8%	19.7%
Improvement vs. Module-per-CU	1.0	1.8x	2.6x	5.1x
Improvement vs. One Module per Package	0.6x (*i.e. 1.8x worse!*)	1.0	1.5x	2.9x

Table 3.5: Public interface sizes in Electric 8.0.1

has on static analysis and user effort. We'll consider four cases:

One Module Per Compilation Unit: Each of Electric's approximately 650 compilation units is its own module. Any entity not declared `private` is potentially referenced from some other compilation unit. For this case, we use the count of non-`private` entities as the total size of the `public` interfaces of all modules.

One Module Per Package: Each Java package is treated as a module. Java `private` and "package private" (*i.e.* no modifier) entities cannot be statically referenced from other modules. For this case, we used module analysis to discover how many of the potentially referenced entities are actually referenced across module (that is, package) boundaries. The number of such entities is the reported total interface size.

Four 'Random' Modules: We created four modules and assigned entire Java packages to them at random. We used module analysis to discover how many potentially referenced entities are actually referenced across module boundaries, and report this as the interface size. The number shown is the average over four random assignments.

Four 'Good' Modules: In this case, we report the numbers from Electric as module-ized and analyzed above. Assignment of packages to modules was based on analysis of the reference patterns in the application.

The data from these four cases is summarized in Table 3.5. From these numbers, we can clearly see that an increase in the size of modules from one module per compilation unit to one module per package yields a significant—nearly 2x—reduction in the number of entities referenced across module boundaries. Increasing module size again to four modules for the entire application—roughly eleven Java packages per module—provides still more reduction. Thoughtfully chosen modules provide nearly twice the benefit of randomly chosen modules, but even randomly chosen modules provide interesting benefit.

The size of the public interfaces of modules is a useful figure of merit because the inference-based analysis techniques used in the thread coloring analysis allow users to avoid annotation of most Java entities that are *not* part of any module's public interface. Thus the 5.1x reduction in the aggregate size of the public interfaces of the modules shown here yields a significant reduction in annotation effort.

Electric differs from other programs the Fluid group has seen in one interesting respect. The `Database` module contains 1731 Java entities that are referenced across module boundaries (out of 1745 total). This represents essentially every `public` type, field and method in the entire module. The explanation for this oddity is that the sole purpose of the `Database` is to define the getters and setters for Electric's underlying data model. This interface is quite large. Table 3.6 on the following page shows the public interface sizes excluding the `Database` in order to remove its unusual size from consideration. The public interfaces of modules for the programs the Fluid group has seen in the field typically fall somewhere between that shown in Table 3.5 and that shown in Table 3.6; that is, we would expect to see the sizes of public interfaces improve more than shown with the Database included, but less than shown with it excluded.

	One Module per CU	One Module per Package	Three 'Random' Modules	Three 'Good' Modules
Non-`private` Entities	11071	11071	11071	11071
Referenced across Module Boundaries	11071	5472	3235	791
% that are Interface	100%	49.4%	29.2%	7.1%
Improvement vs. Module-per-CU	1.0	2.0x	3.4x	14.1x
Improvement vs. One Module per Package	0.5x (*i.e 2x worse!*)	1.0	1.7x	7.1x

Table 3.6: Public interface sizes in Electric *sans* `Database`

Experimental limitations Electric is only one application, albeit a fairly large one. It is reasonable to ask whether annotating other applications would show similar results. Further work with case studies in the field has shown that we can expect results that are similar or better.

3.6 Comparative analysis

There have been many different module systems over the years. This discussion of related work begins with an overview of a variety of closely related systems and their attributes. The section continues with a comparison of the attributes of various module systems. Many of these attributes have been discussed in this chapter; a few appear only in this section. The section closes with a summary of the aspects in which the Fluid module system differs from the others.

3.6.1 Other module systems

Jiazzi

According to its authors, Jiazzi [40] is mostly aimed at supporting mix-ins. It combines classes into a cloud of "atoms." There is no hierarchy of atoms. Atoms hide their internals from other atoms; method conflicts are only possible between methods that are both visible in the same scope, so methods/classes hidden within one atom cannot conflict with methods/classes hidden within another atom. Unlike the Fluid module system, Jiazzi is realized as a Java language extension, and so may require changes to existing source code. Jiazzi also does not work well with existing tools for Java; compilers and Integrated Development Environments are particularly problematic.

MJ

MJ [10] adds static modules to Java, with visibility control. It's much like the Fluid module system, in that it supports general import/export between modules, but MJ has no hierarchy. All modules exist at a single visibility level; users must explicitly import from whatever other module they are after. MJ supports forbidding import into this module from <list-of-other-modules>, forbidding export from this module to <list-of-other-modules>, and permitting export exclusively to <list-of-modules>.

Lag*oo*na

Lag*oo*na [20] is a programming language that is primarily aimed at supporting Component Oriented Programming (COP). Lag*oo*na's modules are both compilation units and the units of deployment. Modules "live' in a flat global namespace; there is no nesting or hierarchy. By convention, Lag*oo*na uses a "hierarchical" naming convention for modules similar to that used in Java. Modules are sealed; they export only the the subset of their contents explicitly marked for export. Import from external modules is also explicit.

MzScheme

MzScheme's [19] modules are the basis for its separate compilation. They also provide namespace and scope management and direct the process of compilation. For the purposes of this comparative analysis, MzScheme's modules are simple compilation units.

Parnas

Parnas introduced systematic thinking about the issues and criteria behind "modularization." The modules he discussed were roughly comparable in size and scope to compilation units in today's languages.

Ada Library Systems

The Ada language standard [52] requires that any attempt to link an inconsistent program (*e.g.* a program containing parts that were not compiled against matching interfaces) must fail. Ada compilation systems prior to GNAT [32] typically accomplished this through the use of a program librarian that tracked compilation dependencies and selected the specific object files to be linked together into a program. Most such Ada Librarians provided make-like functionality. Several also included higher-level system-building capabilities.

The two example Ada systems described below are grouped together with Ada05 in the charts that follow.

Tartan's hierarchical Ada Librarian Tartan Inc.'s Ada compilation system [11] included a Librarian tool that supported a hierarchy of Ada libraries, with inheritance of both code and specifications from libraries higher up the tree. Local compilation could over-ride the versions of compilation units found in higher-level libraries. Each library could have a "spec-library" that contained only those specification files exported to the outside world. All other compilation units were hidden within the library. At system link-time, the user could combine parts from multiple library trees, by choosing one library from each relevant tree as the "link source" for her program. Attentive readers will note the similarity between this design and the Fluid module system.

Verdix/Rational Ada Library import/build paths Each library had an "import path" of other Ada libraries. Specification and implementation files not found locally were searched for up the import path in the typical C-and-Unix-ish fashion. At link time, users designated a group of libraries as the "leaves" of their build; the program was then assembled from the parts found closest to those leaves on each library's import path. Thus, there was no explicit hierarchy or direct control of import/export, but the system was capable of supporting these with clever use of paths, sub-libraries, etc. At least some client sites used the available building blocks to support hierarchical capabilities essentially similar to those provided by the Tartan Ada Librarian.[23]

JSR294 on Superpackages for Java.

JSR294 [36] (in progress as of May, 2008), is a language modification intended to support both information hiding and separate compilation. Currently available documentation addresses only information hiding. The unit of modularity is the `Superpackage`, which has a list of members (which are either packages or nested `Superpackages`) and a list of exported members (which are either types from the current `Superpackage`, or entire nested `Superpackages`). `Superpackages` can contain other `Superpackages` in a tree-like structure; there is no overlap between individual `Superpackages` other than the nesting relationship.

Exporting a type makes both the type itself and all of its public members part of the visible interface of the `Superpackage`. It is not currently possible to export a subset of the public members of a class. Similarly, exporting a `Superpackage` makes its entire visible interface part of the visible interface of the

[23]Author's personal observation.

parent `Superpackage`. It is not currently possible to export a subset of the child `Superpackage`'s public interface.

The visibility restrictions imposed by `Superpackages` will be enforced both by `javac` and by JVMs.

Differences from the Fluid module system:

No selective re-export JSR294 `Superpackage`s can only re-export the entire specification of a child `Superpackage`. Fluid modules can re-export a subset of the interface of a child module.

`Superpackage`s can only export the entire public interface of a type; Fluid modules can be set up to export (or not) on a per-method/field basis. Without selective re-export, `Superpackage`s are unlikely to achieve their stated goals with respect to information hiding.

Module membership granularity `Superpackage`s use the Java package as the unit of module membership; the Fluid module system uses the class. Each decision seems appropriate for the problem at hand: the prototype assurance tool must support overlaying modules onto existing programs; `Superpackage`s are a language change that can mandate changes to user code.

Code in non-leaf modules? JSR294 supports placement of code in non-leaf `Superpackage`s. The Fluid module system does not support placement of code in non-leaf modules. The JSR294 Strawman document [56] explicitly states that they expect that an entire `Superpackage` tree will be processed/compiled in one chunk. The Fluid module system supports one-module-at-a-time analysis, with guaranteed composability. Recall, however, that these two approaches differ by only one small automatic transformation.

Analysis/compilation/deployment granularity JSR294 explicitly anticipates that an entire `Superpackage` hierarchy (from a top-level `Superpackage` down) will be present at deployment time; they strongly imply that this property will hold for compile/analysis-time as well. This seems like an excellent idea for deployment, but appears to be deficient for analysis. In particular, they appear to require whole-hierarchy analysis (rather than whole-module analysis as in Fluid). This larger analysis chunk-size could be a performance/scalability issue.

It seems likely that the compile-time part of this granularity assumption is a consequence of allowing code in non-leaf `Superpackage`s. If JSR294 chose to either adopt our model (code only at the leaves) or to internally transform to that model, they could easily permit module-at-a-time analysis.

Vault

Vault [51] is a C-like programming language from Microsoft Research, aimed at providing much-improved support for reliable programs. Vault supports analysis of and reasoning about the usage rules for APIs. Vault also defines a module system with explicit interfaces and controlled import and export of functions and data.

Vault's modules are tree-structured in that modules can be nested within other modules. However, top-level modules are the basic unit of separate compilation, so nested modules cannot be compiled separately from their parent. One implication of this is that Vault tends to be a small-module system.

Vault modules can define `Interface`s; in addition, `Interface`s can be *ascribed* to modules. A module that has been ascribed with an interface implements that interface just as with interface implementation in Java.

CMOD

The CMOD system [55] formalizes type-safe information hiding and separate compilation for C, guaranteeing—among other things—that .h and .c files actually agree on function and variable types. Addition of these features to C provides a type-safe small-module system approximately comparable to those of other languages like Modula, Ada, and Java. CMOD does not add inheritance, interfaces, or templates to C; those features, while interesting and important, are not fundamental to module systems.

3.6.2 Module system attributes

We structure this analysis using several criterion matrices. The criteria shown here are refinements of those discussed in Section 3.3.2 on page 40. The mapping from this section's attributes to the criteria of Section

	Module hierarchy style	Identified interfaces	Export control		Import control	
			Basic	Complex	Basic	Complex
Fluid	Tree*	Yes	Yes	Yes	Yes*	Yes
Jiazzi	Flat	Yes	Yes	N/A	N/A	N/A
MJ	Flat	Yes	Yes	No*	Yes	No
Lagoona	Flat	Yes	Yes	?	N/A	N/A
MzScheme	Tree	Yes	Yes	?	N/A	N/A
Parnas	N/A	Yes	Yes	N/A	N/A	N/A
Ada05	Tree	Yes	Yes	No	Yes	No
JSR294	Tree	Yes	Yes	?	Yes	?
Vault	Flat*	Yes	Yes	No	N/A	N/A
CMOD	Flat	Yes	Yes	?	?	No

Table 3.7: Module system visibility control attributes

3.3.2 is given in the description of each attribute.

The meaning of each of the attributes in Tables 3.7 through 3.10 is explained below. Table entries with asterisks are explained either in the text description of the attribute or in the module-system-specific text. Question mark entries are used when the available papers do not provide sufficient detail to determine the choices made by that module system.

Visibility control attributes

Table 3.7 compares the module systems' support for various forms of visibility control.

Module Hierarchy Can modules nest or overlap? If so, is there a hierarchy of modules? What shape is that hierarchy? The entry for Fluid is "Tree*" because programmers can override the default tree-shaped hierarchy to provide more complex visibility when needed. The entry for Vault is "Flat*" because Vault permits tree structured modules. Even so, top-level modules are Vault's compilation units; all child modules in the hierarchy below a top-level module are compiled together with it.

Basic export control The issue here is whether a module can restrict the visibility of its own contents to the outside world, effectively supporting information hiding. It is difficult to imagine a useful module system that does not support this attribute.

Basic import control Can a module selectively restrict the importation of visible items from elsewhere, thus preventing internal units from depending on those visible items? This question is relevant only for those module systems that support large modules. The "Yes*" entry for the Fluid module system indicates that the design includes this feature, but the prototype assurance tool does not implement it.

Complex export control Can a module export some of its own contents to other modules generally, while simultaneously denying access to some specific subset of other modules? As an example (with made-up syntax): "export foo, but not to untrusted_module". Also, can a module export some of its contents to a specific subset of other modules that would not ordinarily have visibility to those contents? The "No*" entry for MJ indicates that MJ supports only explicit import and export control; there is no default visibility at all.

Complex import control Can a module import ordinarily visible items while simultaneously forbidding the import of any items from an untrusted module? This attribute is relevant only for those module systems that support large modules.

Backwards compatibility

Table 3.8 on the next page compares the module systems' attributes relating to compatibility with existing code.

	Stand-off module definition	Requires language changes	Requires source code changes
Fluid	Permit	No	No*
Jiazzi	No	Yes	Yes
MJ	Require	No	?
Lagoona	No	No	?
MzScheme	No	No	?
Parnas	No	Yes	N/A
Ada05	No	No	No
JSR294	Require	Yes*	No
Vault	No	No	No
CMOD	No	No	No

Table 3.8: Module system support for existing code

	Large modules	Supports composable analyses	Supports incremental adoption
Fluid	Yes	Yes	Yes
Jiazzi	No	No	No
MJ	Yes	No*	Yes
Lagoona	No	No	?
MzScheme	No	No	No
Parnas	No	Yes*	?
Ada05	No*	Yes*	Yes
JSR294	Yes	Yes	?
Vault	No	Yes	No
CMOD	No	Yes	?

Table 3.9: Module system scalability attributes

Stand-off module definition Is it possible to group compilation units into modules without modifying the source text of some or all of the compilation units? Some systems—typically those whose modules are built-in to a programming language—do not permit this. Others mandate it (*e.g.* MJ). Yet others (*e.g.* Fluid) support both modes of operation.

Language Changes Does implementing the module system require changes to the language for which it is designed? "No" entries typically indicate that the module system either uses stand-off annotation; puts its annotations in comments; or is built-in to the underlying language. "Yes" entries are typical of systems that are extending an existing language to create a new language that has a built-in concept of module. The "Yes*" entry for JSR294 recognizes that the JSR is a change to the Java language currently making its way through the Java Community Process; thus JSR294 will eventually become *part of* Java, at which point it will no longer be a language change.

Source Code Changes Does the act of gathering code into a module require changes to the source code being so grouped? Answers of "No" appear either because of some form of stand-off definition of modules, or because the module system is a mandatory part of the source language. Answers of "Yes" typically indicate that the module specification is expressed directly in the source code in some fashion. The Fluid module system permits the expression of module-ization annotations either in the source code or in a stand-off annotation file; thus its "No*" entry.

Scalability attributes

Table 3.9 compares the module systems using attributes related to their support for scaling to large programs.

	Sealed modules	Static or Dynamic modules	Static linking	Dynamic linking	Duplicated contents	Typed modules	Mixin / Inheritance support at Module level	Validation
Fluid	Yes*	Static	Yes	Yes	No	No	No	Yes
Jiazzi	No	Static	?	?	No	Yes	Yes	Yes
MJ	Yes	Both	Yes	Yes	Yes	No	No	Yes
Lagoona	?	?	?	?	No	No	Messages instead	Yes
MzScheme	?	Both	Yes	Yes	No	Yes	Yes	Yes
Parnas	Yes	Static	Yes	?	No	?	No	?
Ada05	Yes	Static	Yes	Yes	No	Yes*	Yes	Yes
JSR294	Yes	Static	Yes	Yes	No	No	No	Yes
Vault	Yes	Static	Yes	Yes	No	Yes	via Interfaces	Yes
CMOD	No	Static	Yes	Yes	No	No	No	Yes

Table 3.10: Other module system attributes

Large modules This attribute describes the number of compilation units typically found in a module. Module systems that support large modules generally support arbitrarily large numbers of compilation units per module, and certainly anticipate at least hundreds or thousands. Module systems that do not support large modules generally either define modules as being the same as compilation units, such as Parnas' original modules paper, or support at most a few compilation units per module. An example of the latter is the Ada05 package system. Although this can, in theory, support arbitrary numbers of child packages per package, in practice no experienced programmer uses more than a few.

Supports composable analyses Does the module system support module-at-a-time analysis of code (either by a compiler or other analysis tool)? An answer of "yes" typically requires either sealed modules, a concept of "module" built-in to the language, or that "module" and "compilation unit" be one and the same (the Yes* entry for Parnas' modules). MJ supports whole-module analyses inside sealed modules, but provides no support for composing these analyses to cover an entire program, thus receiving a "No*" entry. The "Yes*" for Ada05 indicates that the language allows implementors to support separate compilation of a package and its children (which would result in a No), simultaneous compilation of a package and its children (resulting in a Yes), or any combination.

Supports incremental adoption Does the module system support incremental addition of modules to existing programs? This attribute is most interesting for module systems that either are not part of their underlying source language (such as Fluid and MJ) or that are optional parts of their underlying language (as in Ada05). When the module system is a mandatory part of the source language, modules cannot be added to preexisting code in an incremental fashion—the language required their use from the start. The "?" for JSR294 reflects the lack of publicly available information on its details.

Other module attributes

Sealed Modules Does the module system support or require sealed modules? A sealed module is one whose entire contents are known statically at compile/analysis time, and whose contents cannot be changed later at runtime. For example, MJ supports both sealed and unsealed modules; Fluid assumes that modules are sealed, but the prototype implementation neither checks nor enforces this assumption; Jiazzi does not even have the concept in the first place.

Static/Dynamic Modules Does the module system support modules with statically known membership?

Dynamically changing membership? Both?

Static Linking Does the module system support statically linked programs? Note that some systems support both static and dynamic linking.

Dynamic Linking Does the module system support dynamically linked programs?

Duplicated Contents Is it possible for a single compilation unit to be a part of more than one module? If so, must the compilation unit be of a single system-wide consistent version? Systems whose answer is "no" require that each compilation unit be part of at most one (smallest enclosing) module. Systems whose answer is "yes" may or may not additionally support modules with different versions of the same compilation unit.

Typed Modules Are modules first class entities in the type system? This question is typically answered "yes" only for languages that include a built-in notion of module. The entry for Ada05 is "Yes*" because Ada's generic packages are typed, but are not first-class types in the usual sense.

Mixin/Inheritance support at module level Jiazzi's modules fundamentally exist to support mixin and inheritance. Lagoona and Vault provide this capability through the use of messages and interfaces, respectively.

Validation Does the module system perform or mandate a consistency check to assure that the as-written code conforms with the modules specification? It is difficult to imagine a useful module system that does not have this attribute.

3.6.3 What's different about the Fluid module system?

The Fluid module system:

- supports a hierarchy of modules, with default scoping of import and export. It also permits violating the default hierarchy when necessary.
- enables improved tool support, especially with respect to separable analysis. Other module systems have supported composable analyses across modules in the past, but only for compilation units as modules. The Fluid module system both supports a larger granularity of analysis and also supports cross-module composition of analyses.
- supports the use of module boundaries and interfaces as analysis cut-points. That is, it allows other analyses to require annotation of module APIs so that they can infer annotations inside the module to reduce user effort.
- guarantees that a scalable composable analysis can process the contents of each module individually. There is never a need to load the contents of more than one module at a time. This property is ordinary for flat module systems, but appears to be unusual among module systems that support a hierarchy of modules.
- sits in a previously-empty spot in the multi-dimensional feature-space of module systems.

3.7 Future Work

To ensure the sealed module property needed for module-at-a-time inference, the Fluid group should implement a trusted class loader for dynamically loaded code. This loader must check that the loaded classes are (a) the specific classes that were analyzed statically, (b) are being loaded "within the context" of the module that statically "owns" them, and (c) that the modules accessed/invoked via cross-module references are in fact the ones that we analyzed statically (or are sufficiently compatible that all analysis results will still hold). Property (a) is clearly feasible; properties (b) and (c) will require rather more work, but should be manageable with reasonable engineering effort.

To achieve the extremely low numbers of required annotations described in Section 3.5.2 on page 62, the Fluid group must extend support for scoped promises to allow wild-card matching across all packages within a module. This is a straight-forward engineering effort; the only tricky issue is coming up with a file or "place" that represents the module scope in order that we may place scoped promises there.

Some of the annotations described above are not yet fully implemented (notably the "**`@ForbidImport`** and **`@Export ... to ...`**" annotations). Implementing these features is straight-forward engineering; they do not impose any additional research risk. Further, the prototype assurance tool should be extended to support automatically moving code from non-leaf modules into newly-constructed leaf modules; this would provide the appearance of code in non-leaf modules while maintaining the capability for module-at-a-time analysis. Finally, the prototype assurance tool could provide a visibility annotation wizard to assist users with annotation of their existing programs. It could also support queries as described in Section 3.3.2 on page 40.

Combining the Fluid module system with Halloran's layers concept (*i.e.* nesting of modules within layers or of layers within modules) is not yet supported. Again, this is straight-forward engineering that imposes no additional research risk.

Chapter 4

Case studies

4.1 Introduction

This chapter serves as the validation section of my dissertation. In it I present detailed results from six case studies, along with discussion of the challenges posed by and the lessons learned from each. The case studies show how prevalent thread usage errors are; even example Applets like GraphLayout have one. The primary lesson of this chapter is that the thread coloring analysis can express realistic threading models, and that the prototype assurance tool can check the models' consistency with as-written code, thus producing interesting and useful results. Further, the tool successfully handles large-scale programs and complicated threading models. The final lesson is that the prototype tool has significant deficiencies in reporting results; additional work in this area would be required to produce a product-quality tool.

It is important to note that case studies—as performed by the Fluid group—are not formal user studies. During these studies we investigate the analysis and its implementation, not the users of the analysis. The specific code examined in our in-field case studies is usually not available in advance, nor is it selected by the Fluid group. Rather, the host organization selects the projects and programs; Fluid group members see the code for the first time on arrival for the case study. Developers who participate in case studies in the field are there to get our help with specific issues in their source code. The Fluid group provides this help by using our analyses in collaboration with the developers. Developer reactions described here are purely anecdotal, and were volunteered by those developers.

I performed some of the case studies at Carnegie Mellon, and others in the field in cooperation with a variety of third-party developers. One case study—Electric—involved significant advance preparation at Carnegie Mellon, a several-day study in the field, and additional work back at the university after the fact. Field work is particularly useful for investigating questions involving developer opinions, ROI, limited time and training, and engagement with other people's code. Lab work provides time for deeper investigation of modeling patterns and for investigating issues of scale and the effort needed to achieve it; however, lab work provides a lesser degree of engagement with other people's code.

As a member of the Fluid group, I have performed additional case studies—both in the field and in the laboratory—that are not described here. Some of the field studies are omitted for reasons of confidentiality. I have also omitted those laboratory studies that fail to add significantly to the evaluation of this work. My experience with the omitted studies is consistent with the case studies reported here.

In the next section I discuss the aspects of realism and practicability that I intended to address through these case studies. In the remaining sections I discuss the individual studies. Each case study begins with a description of the issues addressed in that study, followed by a brief description of the application being analyzed. The details of each study are presented in narrative form; this form demonstrates the typical engagement of user, analysis tool and code being analyzed. The narrative is interrupted by occasional "Analysis" sections, which highlight important results or issues with the tool or the analysis algorithms.

The chapter closes with a summary of the key results of the case studies.

4.2 Realism and practicability

I undertook these case studies for two reasons: first to discover whether my analysis and its implementation function at all, and second to assess realism and practicability. The aspects of realism and practicability that concerned me include:

Does it work at scale? Does the analysis handle medium-sized programs—100K SLOC or so—with ease, or is it limited to small programs? Can the analysis scale to large programs? These questions relate both to the analysis in the abstract and to my experimental implementation of the analysis. I have demonstrated successful use of thread coloring on a 140KSLOC program (see Section 4.5 on page 90); module-at-a-time analysis enables scaling to much larger programs.

What do "real developers" think? I gathered anecdotal indications of the reactions of real developers to the analysis, to the models it uses and their expression, and to the results it provides. A number of developers have indicated their desire to continue using thread coloring analysis and the prototype assurance tool. The results of case studies in the field suggest that a "product-ized" version of the thread coloring analysis and tool would be well received.

Can it handle other people's code? Testing on carefully chosen small examples can provide "laboratory" indications of the properties of the analysis. But how does it perform on real production code with all its quirks and warts? Studying the analysis using other people's code also helped me avoid the natural tendency to look more closely at best-case results. Case study results show that the prototype implementation of thread coloring handles real code successfully.

Can we produce results with limited time and training? Laboratory studies performed by the analysis developer provide a useful indication of results that are achievable with great quantities of time and deep understanding of the analysis. What sort of results can we provide in a limited time, with little or no user training? My case study experience shows that sophisticated users can write their own design intent with less than one hour of training. Less experienced users needed help writing design intent, but could easily understand design intent once it had been written. However, both kinds of user required assistance navigating and interpreting the results produced by the prototype tool.

Are the analysis model, design intent, and key abstractions "realistic enough?" Does the analysis capture the kinds of results I intended to find? Can developers express the necessary design intent without undue effort? Are the simplifying assumptions I have made close enough to reality to allow the analysis to operate on real code? The case studies show that the prototype thread coloring analysis and its annotations have been sufficient to express all threading models we have encountered so far. The simplifying assumptions have not been a problem in practice.

What about "Team Return On Investment (ROI)" and "Developer ROI?" Does the analysis, along with its experimental implementation, provide development teams with sufficient return on investment to hold their interest? That is, would a team be willing to allocate developer time to use this technique?[1] Independent of the team's return on investment, does the analysis provide sufficient return to the *individual developer* to make her willing to use it? Developers in the field are busy people, with jobs to do and deadlines to meet; their individual incentives often differ from those of the organization. During case studies in the field, this issue manifested itself as "Provide an interesting result before lunch on the first day, or the developers will evaporate." The Fluid group has met this challenge successfully for every case study to date. However, because I helped the third-party developers write annotations and operate the prototype assurance tool during each case study in the field, I lack information about end-user experience in the absence of my expertise.

4.3 GraphLayout case study

In the GraphLayout case study I will show how to express the standard threading policy imposed upon clients of the AWT GUI framework, show inheritance of constraints from framework code and how it reduces model expression effort. I further show that the thread coloring analysis detects calls made from the wrong thread.

[1] The terms "Team ROI" and "Developer ROI" were suggested by Dr. William Scherlis.

Case Study	Size in KSLOC	Application Information	Attributes Explored	Lessons Learned
Graph Layout (lab)	0.4	One of Sun's standard demonstration Applets	Viability	Thread coloring is viable. Incremental adoption and annotation inheritance are really necessary for practicability.
SkyViewCafe (lab)	25	Astronomy Applet & application. Notoriously diligent & careful author	Model Expressiveness, Scalability	Thread usage policies change. Even very careful authors don't "get" the full implications of those changes Color constraint inference required
JHotDraw (lab)	~60	Open-source graphic editor	Incremental adoption, model expressiveness, usefulness of localized color environment information	Reverse-engineering thread usage policy may be impossible without developer knowledge Changes in AWT/Swing threading rules not well-known
JPL Chill GDS (field)	~100	Space Science ground data system. Manages acquisition of and access to downloaded science data. Soft realtime.	Model SWT GUI Use data coloring to demonstrate single-thread access for data	Modeling SWT easy. Data coloring works. Proved single-thread access to critical data so the developers removed synch for key data, got (some) speedup, passed regression tests.
JPL "4Thread" (field)	~50	Headless action-planning application for spacecraft and rovers	Model expressiveness, developer opinions, ease-of-use	Expert developer wrote his own annotations after 15min training. Developer had forgotten details of his own thread usage model. Correcting the model tested model expressiveness. No time for data coloring (the goal)
Electric (field & lab)	~140	Open-source VLSI design tool	Developer opinions, model expressiveness, scalability	Composition required for scalability. Modules make a good unit of composition. Annotation effort can be made low.
Nine others (1 NDA lab, 5 field, 3 NDA field)	1 to 1000	Varied, all fielded applications	All	Incremental adoption model is crucial for adoptability. Very low model expression effort is also required for adoptability.

Table 4.1: Summary of case studies performed

```
1  /**
2   * @ColorDeclare AWT, Compute
3   * @IncompatibleColors AWT, Compute
4   * @MaxColorCount AWT 1
5   * ...
6   */
7  package java.awt;
```

Figure 4.1: Thread coloring annotations for AWT

Finally, I show how to use the thread coloring analysis on data to determine which thread roles may touch a data region, then use this information to find a race condition.

GraphLayout was my initial case study. It served to answer the question "Does thread coloring work at all?" and to drive the initial construction of and experimentation with the prototype assurance tool. We used GraphLayout as a running example for the "guided tour" in Chapter 2. The guided tour focused on expression of design intent; this presentation emphasizes the operation of the tool and the results it produced, along with technical detail about GraphLayout itself.

4.3.1 The Application

GraphLayout is a small Java Applet that has been distributed with each version of the Java JDK since version 1.0. It provides an example of how to use the AWT framework to put a window on the screen and interact with objects therein using the mouse, by displaying a graph containing a few nodes and running a relaxation algorithm on them.

4.3.2 Case study details

The threading model

GraphLayout participates in the GUI framework, and so must follow the framework's thread usage policy. For example,

- Do not compute in the GUI event thread (we'll call it the `AWT` thread hereafter)
- Handle GUI events and touch GUI objects *only* in the `AWT` thread
- There is at most one `AWT` Thread (at a time)

To declare and enforce these rules we begin by naming the thread colors we'll use for AWT clients, and later for Swing clients. The thread coloring annotation on line 2 in Figure 4.1 declares the existence of the two colors. We intend that the `AWT` color be used only for the AWT event thread and that the `Compute` color be used for the user's working threads. The annotation on line 3 declares that no thread may ever be simultaneously associated with both the `AWT` and the `Compute` colors. As a consequence of this annotation, we can reason that any thread that has one of these colors certainly does not have the other. The annotation on line 4 declares that there is at most one `AWT` thread at a time. This property is useful for reasoning about sharing of data regions between threads. It is declarative, and is not checked by the analysis.

Initial coloring of code

There are two threads of interest in the GraphLayout Applet: the `AWT` event thread and a thread used to compute the relaxation algorithm for the graph. Figure 4.2 on the next page shows the thread-related sequence of actions when running GraphLayout. Note in particular that the `AWT` thread may invoke callback methods in classes `Graph` and `GraphPanel` even while the `Compute` thread is running the relaxation algorithm.

In Figure 4.3 on page 80 we see some of the initialization and event-handling code for the Applet. As a result of the GUI framework rules, all of the code shown must execute only in the `AWT` thread. Specifically, the default constructor must run in the `AWT` thread because it touches GUI objects and invokes GUI-only methods (the constructors for `Button` and `Checkbox` at lines 13 and 15. The `init` method (at line 17) is a callback that is part of the `Applet` framework. By the nature of that framework it will certainly be invoked in

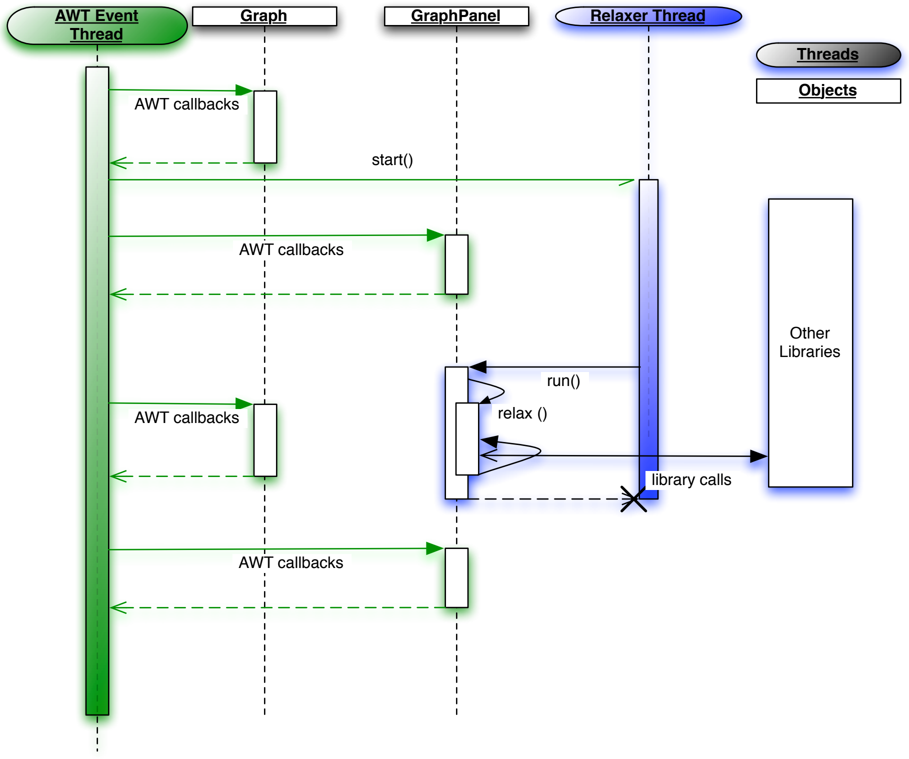


Figure 4.2: Pseudo-sequence diagram for GraphLayout

the GUI event thread. The `init()` method is also constrained to run in the `AWT` thread both because it calls GUI-only methods and because the method it overrides is similarly constrained[2]. No annotation is required because this constraint is inherited from the annotation attached to the framework's definition of the method. Finally, `actionPerformed()` (at line 26) is a callback from the AWT framework (shared with the Swing framework). The framework will certainly invoke it in the `AWT` thread. By the framework's rules it should never be invoked in any other thread. Again, the necessary annotation is inherited from the framework.

In Figure 4.4 on page 81 we see excerpts of class `GraphPanel`. The `run()` method is invoked by Java library code after the `Compute` thread is started. It repeatedly calls the `relax()` method, does some additional work, and then sleeps to reduce the change rate of the graph display. The `relax` method uses a relaxation algorithm to move the nodes around, and then calls `repaint()` to tell the GUI that the graph has changed and should be redrawn.

The `run()` method in `GraphPanel` is invoked as the main entry point of a compute thread in accordance with the rules of the Java library's `Thread` and `Runnable` frameworks. We note this by annotating the method with a matching color constraint (line 15). We use a `@Color` annotation rather than placing a `@Grant` inside the method in order to indicate that user code must avoid direct invocation of the `run()` method. We avoid annotating the `relax()` method at all (line 33), because it is neither part of the visible interface of the `GraphLayoutApp` module, nor is it an entry-point such as `run()`. Instead, we expect that the analysis tool's inference algorithm will compute a suitable color constraint for `relax()`.

At this point, when we run the thread coloring analysis we discover that GraphLayout is a well-behaved client of the AWT with respect to thread usage and code. Figure 4.5 on page 82 summarizes the thread coloring results.

[2]Strictly speaking, substitutability is maintained as long as it is *capable* of running in the `AWT` thread; it could permit other threads to execute it as well. In this case, however, contents of the method preclude that second option.

```
/** @Module GraphLayoutApp */
package GLPack;
...
/** @ColorImport java.awt.*;
 * @Promise "@Color AWT" for new()
 * @Region Controls
 * @MapFields controlPanel, scramble, ... random into Controls
 */
public class Graph extends Applet implements ActionListener, ...
{
  GraphPanel panel;
  Panel controlPanel;
  Button scramble = new Button("Scramble");
  ...
  Checkbox random = new Checkbox("Random");

  public void init() {
    {
      setLayout(new BorderLayout());
      panel = new GraphPanel(this);
      add("Center", panel);
      controlPanel = new Panel();
      add("South", controlPanel);
      ...
    }   }
  public void actionPerformed(ActionEvent e) {
    Object src = e.getSource();
    if (src == scramble) {
      for (...) {
        Node n = panel.nodes[i];
        if (!n.fixed) {
           n.x = ...;
           n.y = ...;
        }
      }
      return;
    }
...
} ... }
```

Figure 4.3: Initialization and event-handling code from GraphLayout

```
/**@Module GraphLayoutApp */
package GLPack;
...
class GraphPanel extends Panel implements Runnable
...
{ ...
  //@Aggregate Instance into Instance
  Node nodes[] = new Node[100];
  ...
  //@Aggregate Instance into Instance
  Edge edges[] = new Edge[200];

  Thread relaxer;
  ...
  /** @Color Compute; */
  public void run() {
    Thread me = Thread.currentThread();
    while (relaxer == me) {
      relax();
      if (...) {
        Node n = nodes[...];
        if (!n.fixed) {n.x += ...; }
      ...
      }
      //update(offgraphics);//bogus call to update!
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {
        break;
      }
    }}

  synchronized void relax() {
    for (int i = 0; i < nedges; i++) {
      Edge e = edges[i];
      ... = nodes[e.to].x - ...;
      ...
      nodes[e.to].dx += ...;
      ...
    }
    ...
    repaint();
  }
  public synchronized void update(...) {
    ...
    int x1 = (int) nodes[e.from].x;
    ...
  }
  ...
}
```

Figure 4.4: Selected code from `GraphPanel.java`

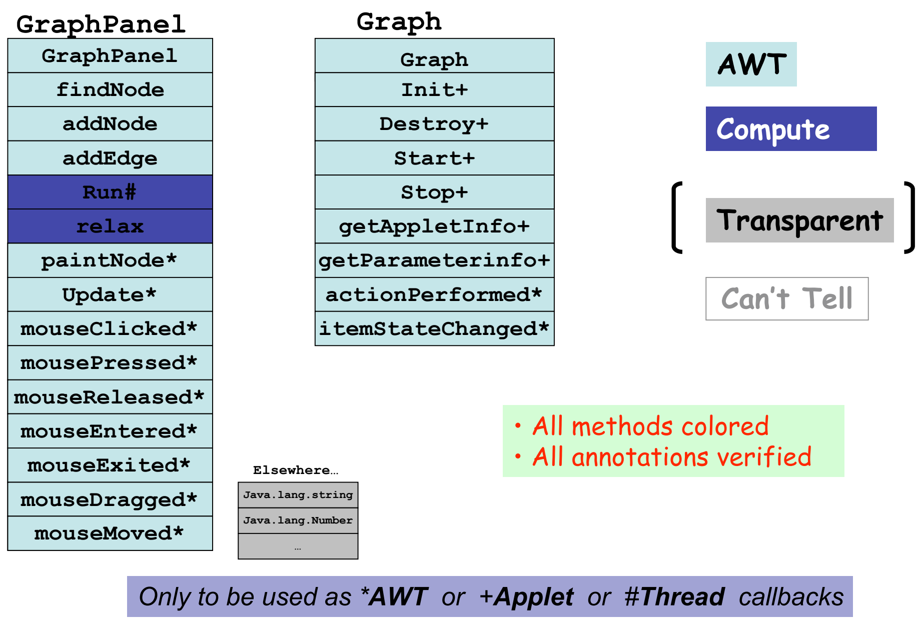
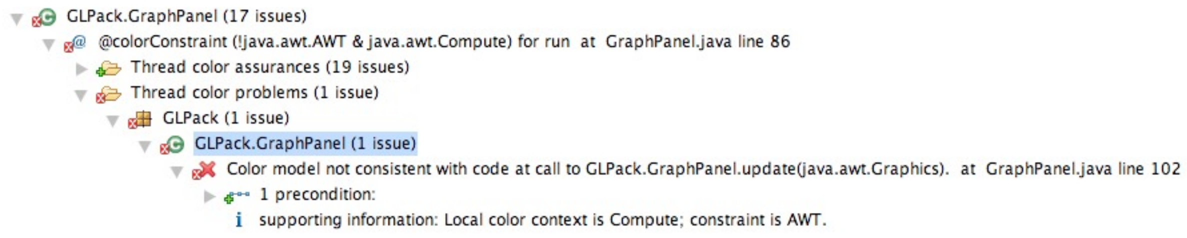


Figure 4.5: GraphLayout coloring results



```
GLPack.GraphPanel (17 issues)
    @colorConstraint (!java.awt.AWT & java.awt.Compute) for run  at  GraphPanel.java line 86
        Thread color assurances (19 issues)
        Thread color problems (1 issue)
            GLPack (1 issue)
                GLPack.GraphPanel (1 issue)
                    Color model not consistent with code at call to GLPack.GraphPanel.update(java.awt.Graphics).  at  GraphPanel.java line 102
                        1 precondition:
                        supporting information: Local color context is Compute; constraint is AWT.
```

Figure 4.6: Error for bogus call to `update()`

```
1  /**
2   * @Region TheGraph
3   * @MapFields nnodes, nodes, nedges, edges into TheGraph
4   *
5   * @ColorizedRegion TheGraph
6   */
7  class GraphPanel extends Panel implements Runnable, MouseListener ... {
8    ...
9  }
```

Figure 4.7: Data coloring annotations for `GraphPanel`

Introducing an error

We now demonstrate detection of a call from the `Compute` thread to a method that should be invoked only from the `AWT` thread. On line 25 of Figure 4.4 on page 81 there is a commented-out call to `update()`. A naive user might place such a call in an attempt to force repainting of the display after he is done modifying the model. If we uncomment this line and run the thread coloring analysis again, we see that the tool detects the attempt to invoke `update()` from the wrong thread. Figure 4.6 is a screen shot showing the resulting error message. The bottom three lines of the figure show the error and its supporting details. First, the message tells us that the color model and the code do not match. It also identifies the method being called and the location of the call site. The last line shows that a current color environment of `Compute` does not satisfy a color constraint of `AWT`.

Coloring data

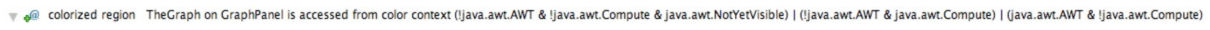
We now turn to using thread coloring to determine whether accesses to data are guaranteed to be single-threaded or are potentially multi-threaded. Single-threaded access need not be locked, as there is no possibility of a race condition. Potentially multi-threaded access to data indicates that some kind of synchronization discipline should be used. For this example we will assign some of the data fields in GraphLayout's graph representation to a data region[3] and then ask the prototype assurance tool what thread colors access that region.

Line 2 of Figure 4.7 shows the `@Region` annotation that declares the existence of region `TheGraph`. The `@MapFields` annotation on line 3 says that the `nnodes`, `nodes`, `nedges` and `edges` fields of every `GraphPanel` object are part of region `TheGraph`. The `@ColorizedRegion` annotation on line 5 tells

[3]Data regions (due to Greenhouse & Boyland [24]) are identified chunks of memory that contain objects or portions of objects. My analysis uses Greenhouse/Boyland effects analysis to determine which regions are accessed by the various memory references in a program.

```
class Node {
  //@MapInto GraphPanel#TheGraph
  double x;
...
```

Figure 4.8: Data coloring annotations for `Node`



```
colorized region   TheGraph on GraphPanel is accessed from color context (!java.awt.AWT & !java.awt.Compute & java.awt.NotYetVisible) | (!java.awt.AWT & java.awt.Compute) | (java.awt.AWT & !java.awt.Compute)
```

Figure 4.9: Data coloring result for `TheGraph`

the analysis tool to compute an expression that indicates which combinations of colors are associated with the current thread at all of the references to any data in region `TheGraph`. Finally, the `@MapInto` annotation on line 2 of Figure 4.8 says that the `x` field of every `Node` object is also part of region `TheGraph`.[4]

Figure 4.9—which reads "Colorized region TheGraph on GraphPanel is accessed from color environment (!java.awt.AWT & !java.awt.Compute & java.awt.NotYetVisible) | (!java.awt.AWT & java.awt.Compute) | (java.awt.AWT & !java.awt.Compute)"—shows the message produced by the prototype assurance tool. Data region `TheGraph` is accessed from code that is colored `Compute` and from code that is colored `AWT`. Recall that we declared these thread colors as being incompatible. Whatever specific thread has the `Compute` color at any given instant is clearly distinct from a thread that has the `AWT` color at the same instant; we have failed to prove single-threaded-ness. Accordingly, we can reason that the data in `TheGraph` is potentially accessed from more than one thread. `TheGraph` is a candidate for some form of synchronization or locking analysis. Because GraphLayout is very small, we will use inspection rather than a more formal analysis.

Note that the result shown in Figure 4.9 is marked neither as an error nor as a warning. This is a consequence of the `@ColorizedRegion` annotation, which acts as a query rather than as a declaration. The computed result is just that: a result. There is no assertion here to be proven true or false.

Armed with the knowledge that data in the graph may be accessed from more than one thread, we now return to the source code. On line 33 of Figure 4.4 we see that the `relax()` method is declared to be synchronized. This suggests that the authors of GraphLayout considered the question of multi-threaded access. It appears that this synchronized keyword, along with the matching synchronization on the `update()` method in class `Graph`, (line 44 of Figure 4.4) is intended to protect against a race between the `AWT` thread and the `Compute` thread when reading and writing the data of the underlying `Graph` objects. However this synchronization is not sufficient. By inspection we can see that `GraphPanel.run()` both reads and writes data in the graph outside any synchronized block (line 22 of Figure 4.4 on page 81). Recall that this method was found to be part of the `Compute` thread. Similarly, `actionPerformed()`—which is also outside the synchronized block, and guaranteed to run only in the `AWT` thread—can also read and write data in the graph (see lines 31 and 32 of Figure 4.3 on page 80). We have found a race condition.

The consequence of this race condition is not severe in itself. At worst a node may be disconnected from one of its edges for a single update of the display of the graph. The larger consequence of this error comes from the common use of this Applet as a concrete pattern. Fairbanks has found [13] that bugs in widely copied examples frequently propagate into code derived from those examples, and that the errors persist long after the example code is corrected.

4.4 Sky View Café case study

In this case study we will see the use of thread coloring on a very heavily multithreaded application containing twenty different thread roles, with multiple instances of each role. I also show how to model a `Runnable` that may be called from either the `AWT` event thread or from its own background worker thread—a pattern left over from before anonymous inner classes actually worked on most platforms. In addition we use thread colors to successfully diagnose a violation of the AWT/Swing thread usage policy that could cause state consistency errors or safe publication errors.

This laboratory case study was originally undertaken to study an application that was larger than a toy example, but not a significant scalability challenge – Sky View Café is about 25KSLOC. It additionally turned out to challenge my model expression techniques, and provided an early test of scalability in terms of numbers of thread colors. Another challenge in the study turned out to be reverse-engineering the developer's intended thread usage patterns.

Repeat items encountered in this case study (not highlighted below):

- AWT/Swing GUI threading pattern

[4]A more complete investigation would involve mapping all data that is part of the graph into region `TheGraph`. I omit the other fields in the cause of brevity.

Package	Colors Needed
`util`	ProcessMonitor
`skyviewcafe`	AtlasQuery, AtlasQueryDialog, KeyRepeater, DigitWarningTimer, InsolationRenderer, InsolationViewRepainter, IPLocator, TimeTracker, TabViewTableGen, MapShadower, MIGetter, Blinker, PDGetBase, SaveThread
`gui`	ImageRetriever, MessageMarquee, MiniAdjusterRepeater, PictureButtonRepeater, PopupPoller

Table 4.2: `Compute` thread colors in Sky View Café

- Declaration of thread colors
- Incompatible thread colors
- Introducing errors to see tool error checking behavior

4.4.1 The application

Sky View Café [54] (SVC hereafter), written by Kerry Shetline, is an astronomy application that can show views of the sky from any place in the Solar System at any time of the user's choice. It runs both as an application and as a Java Applet inside a web browser or other container. SVC was originally written using the AWT GUI framework and Java version 1.1. It has been updated repeatedly over the years, and is now written using Java version 1.4. This long history, however, means that SVC contains code patterns that pre-date many current Java language features (*e.g.* working anonymous inner classes). The current version of SVC supports JVMs back to version 1.3, and contains work-around code for old browsers and JVMs to maximize compatibility.

4.4.2 Case study details

We begin our case study with a survey of `Threads` and `Runnables`, in order to determine what thread colors we will need. SVC is very heavily multi-threaded. We quickly find that, in addition to the AWT/Swing event thread, there are threads to render each individual animated view, I/O thread for slow operations—Internet access, ephemeris database access, etc.—and quite a few timer threads. These latter are used both for animation timing and also to work around key-repeat bugs in some browsers. At the end of this survey we have gathered twenty different roles for threads, as shown in Table 4.2. The names of the colors provide some hint of their intended purpose. Each of these thread colors will be used for threads that are certainly not the AWT/Swing event thread, and that are distinct from the threads represented by the other colors. Examination of the code also shows that each of these colors may be associated with multiple threads simultaneously. We record these properties by declaring the colors, noting that they are all incompatible with each other and with the `AWT` thread (using a single `@IncompatibleColors` annotation that names all twenty colors along with the `AWT` color), and asserting unknown multiplicity for each color.

Modeling a twice-called runnable

SVC invokes an Internet service to estimate the location of the user, given only his IP address. Because the service is external, might fail or might run with speed not controlled by either SVC or the user, this query is made in a background thread. Because of the choice to use a background thread, the main work of SVC can continue even if the location query never completes or returns an error (perhaps because the user is not connected to any network). This is an instance of a thread usage policy members of the Fluid group have seen repeatedly in the field: delegate "less-important" work to another thread, so that slowness or failure does not affect the major features of the system.

SVC uses the `IPLocator` class to implement this location estimating query. Excerpts from `IPLocator` are seen in Figure 4.10. The `IPLocator.run()` method embodies a rather odd programming pattern: it is intentionally invoked twice – once as the main body of the `IPLocator` thread, and a second time by

```
1   public class IPLocator extends Thread {
2     ...
3     /** @Color (AWT | IPLocator) */
4     public void run()   {
5        if (location != null) {//@Revoke IPLocator   //@Grant AWT
6           if (mainPanel != null && !mainPanel.isObserverModified())
7              mainPanel.setAndAddObserver(location);
8           return;
9        }
10       {//@Revoke AWT //@Grant IPLocator
11          SkyObserver    observer = null;
12          try {
13             ...
14             URL              locator = new URL(query);
15             BufferedReader   in =         new BufferedReader(new InputStreamReader(locator.openStream()));
16             ...
17             while ((line = in.readLine()) != null) {
18                if (...) {
19                   // read position in lat/long, etc.
20                   ...
21                   observer = new SkyObserver(longitude , latitude , ...);
22                }
23             }
24             in.close();
25          }
26          ...
27          if (observer != null) {
28             location = observer;
29             SwingUtilities.invokeLater(this);
30          }
31       }
32    }
33  }
```

Figure 4.10: `IPLocator` class from Sky View Café

way of a `SwingUtilities.invokeLater(`**`this`**`)` call. Figure 4.11 on the next page shows the sequence of operations for the `IPLocator` thread and its `run()` method. This pattern, which was common usage in the days before working anonymous inner classes were widely available, presents us with an interesting modeling challenge – how can we arrange to allow the `run()` method to be invoked from either an `IPLocator` environment or an `AWT` environment, while still restricting the actions that may occur inside the method appropriately?

To allow invocation from either color environment, we simply declare that both are acceptable in the color constraint of the `run()` method, as shown on line 3 of Figure 4.10. We now turn our attention to the body of the `run()` method. Examination of the code shows that the code block starting on line 5 will execute only when the location field is set. The only assignment of this field is seen at line 28, just before the call to `SwingUtilities.invokeLater(`**`this`**`)`. We therefore conclude that the block beginning at line 5 executes from the `AWT` event thread. This conclusion is reinforced by the observation that code in the block makes `AWT`-thread-only calls back to the AWT framework.[5] We must adjust the color environment for this block to reflect our knowledge of the current thread. The `@Revoke IPLocator/@Grant AWT` pattern seen on line 5 is the proper idiom for this purpose. Its effect is to remove any possibility of the `IPLocator` color from the current color environment and then to grant the `AWT` thread color. Both of these actions are appropriate because our analysis has just shown that we are actually executing from the `AWT` event thread. As usual, the changes to the current color environment expire at block exit, returning us to the environment of the method as a whole.

We now need to adjust the color environment of the remainder of the method to be `IPLocator` only. To do this, we introduce a block that was not present in the original `run()` method (see line 10) so that we can change the color environment. This block contains all the remaining code in the method, all of which executes solely in the environment of the `IPLocator` thread. The `@Revoke AWT/@Grant IPLocator` annotation pattern seen on line 10 accomplishes exactly this change.

[5]In more modern code this block would be the body of a new `Runnable` declared as an anonymous inner class.

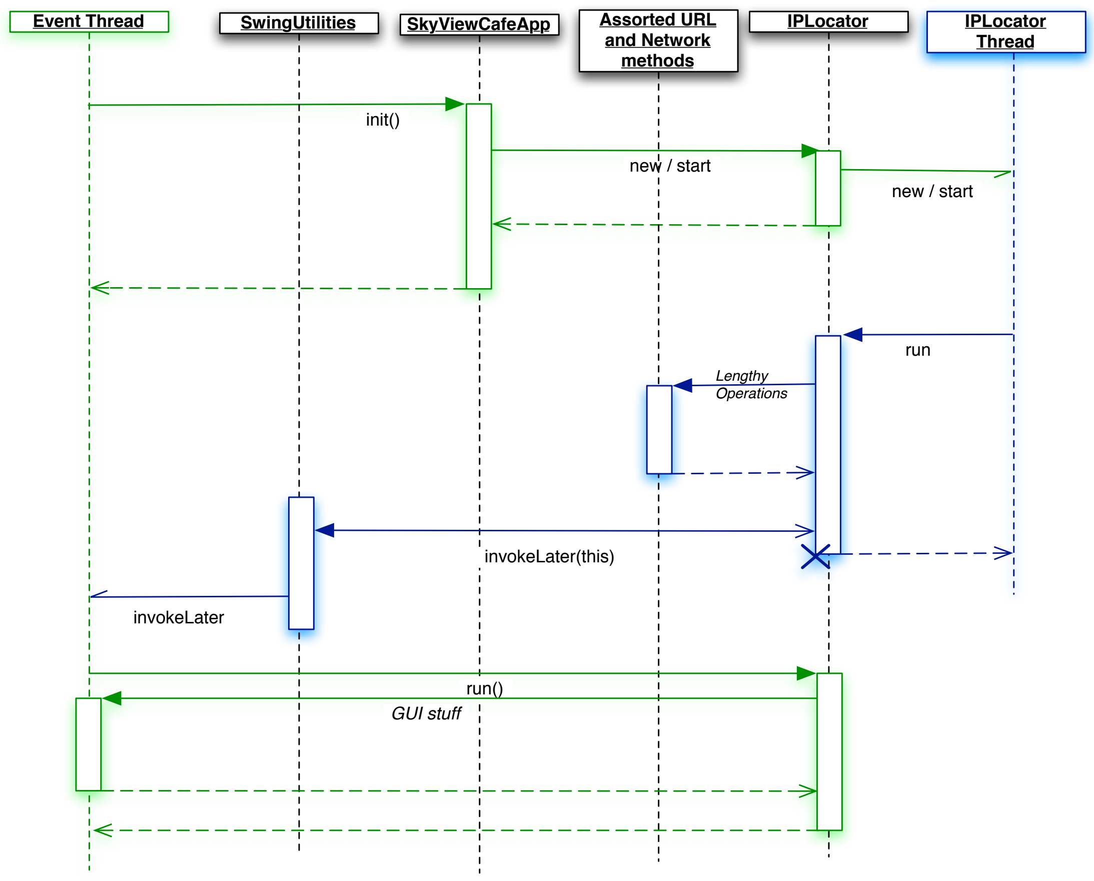


Figure 4.11: `IPLocator` sequence of operations

```
 1  ...
 2  public class TabularView extends GenericView implements ... {
 3   ...
 4   protected void generateTable() {
 5      ...
 6      showButton.setEnabled(false);
 7
 8      // Since we'll be printing off the GUI event dispatch thread, we need
 9      // to extract all relevant info from the GUI first, before starting
10      // the print thread, since it's verboten to access Swing
11      // components, even just to check their state, while not running
12      // in the event dispatch thread.
13      ...
14      new Thread("TabularView.tableGeneration") {
15        /**
16         * @Color TabViewTableGen
17         */
18        public void run()       {
19          int[]   t = mainPanel.getTimeFields();
20          ...
21          SwingUtilities.invokeLater(new Runnable() {
22            /**
23             * @Color AWT;
24             */
25            public void run() {
26              showButton.setEnabled(true);
27            }
28          });
29        }
30      }.start();
31    }
32    ...
33  }
```

Figure 4.12: Excerpts from SVC table-printing code

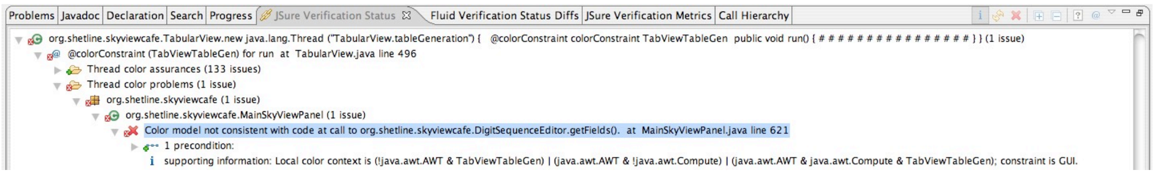
Analysis This twice-called `run()` method is now considered poor practice. Modern code would place the code block starting on line 5 in an anonymous inner class passed to `invokeLater()` as its argument, thus executing that block on the `AWT` thread. The twice-called pattern was the approved way to achieve the same effect before anonymous inner classes were added to Java. This pattern provided an interesting modeling challenge.

The SVC printing thread vs. AWT/Swing thread usage policy

SVC allows users to generate a tabular view of astronomical events, and then print that view. This obviously requires examining data from the Model (c.f. AWT/Swing's Model-View-Controller architecture) and formatting those data for printing. Perhaps less obviously, this formatting should not be executed on the AWT/Swing event thread. If printing takes a long time or fails completely, this could freeze the GUI while the event thread waits for rendering and printing of the tabular data. SVC spawns a new thread to perform the printing actions: the `TabViewTableGen` thread. The usual AWT/Swing thread usage policy prevents this thread from calling GUI methods or touching GUI data, so the printing code in SVC carefully collects the data needed for printing into local variables while running in the `AWT` event thread (and before spawning the `TabViewTableGen` thread) as seen in Figure 4.12.

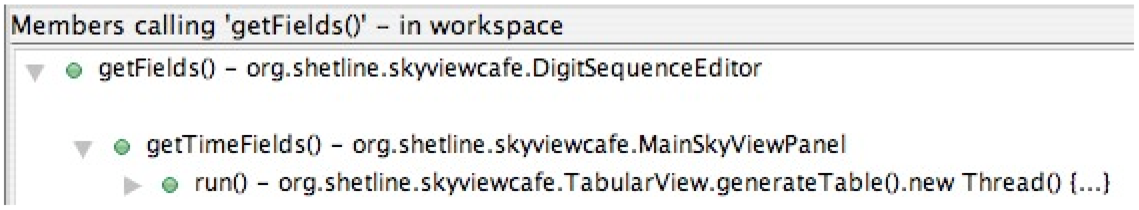
We know from an earlier thread coloring analysis run that method `TabularView.generateTable()` always executes in the `AWT` thread; the inferred color constraint for this method is `@Color AWT`. Lines 8 through 12 of Figure 4.12 contain the developer's comment on the thread usage policy and the necessary actions at this point. Clearly he is aware that his code must not touch GUI items from any thread that is not the `AWT` thread. The "..." at line 13 is the elided code that gathers up information from the GUI and saves it for later use from the `TabViewTableGen` thread. Then, on lines 14 and 30, he creates and starts a new background thread that will actually do the printing. Note that we have declared the `run()` method as being `@Color TabViewTableGen` because it is guaranteed to execute on that thread and should not be invoked from any other thread.

The `run()` method for the `TabViewTableGen` thread begins with a call to


Problems
Javadoc
Declaration
Search
Progress
JSure Verification Status
Fluid Verification Status Diffs
JSure Verification Metrics
Call Hierarchy

```
org.shetline.skyviewcafe.TabularView.new java.lang.Thread ("TabularView.tableGeneration") {   @colorConstraint colorConstraint TabViewTableGen  public void run() { # # # # # # # # # # # # # # # # } } (1 issue)
  @colorConstraint (TabViewTableGen) for run  at  TabularView.java line 496
    Thread color assurances (133 issues)
    Thread color problems (1 issue)
      org.shetline.skyviewcafe (1 issue)
        org.shetline.skyviewcafe.MainSkyViewPanel (1 issue)
          Color model not consistent with code at call to org.shetline.skyviewcafe.DigitSequenceEditor.getFields().  at  MainSkyViewPanel.java line 621
            1 precondition:
            supporting information: Local color context is (!java.awt.AWT & TabViewTableGen) | (java.awt.AWT & !java.awt.Compute) | (java.awt.AWT & java.awt.Compute & TabViewTableGen); constraint is GUI.
```

Figure 4.13: Coloring error in `TabViewTableGen`


Members calling 'getFields()' – in workspace

```
getFields() - org.shetline.skyviewcafe.DigitSequenceEditor
  getTimeFields() - org.shetline.skyviewcafe.MainSkyViewPanel
    run() - org.shetline.skyviewcafe.TabularView.generateTable().new Thread() {...}
```

Figure 4.14: Eclipse call tree for `DigitSequenceEditor.getFields()`

`mainPanel.getTimeFields()` on line 19, then does all the work of printing—represented here by the "..."—and then finishes with an `invokeLater()` call. All of this appears to be just fine, so we run the thread coloring analysis again. Rather to our surprise, the tool reports a thread coloring inconsistency as shown in Figure 4.13[6].

We now investigate the cause of this error. Is our model wrong? Is the code wrong? Our problem must have something to do with the newly granted color `TabViewTableGen` as it is part of the local color environment at the point of error. We examine the static call tree for `DigitSequenceEditor.getFields()`—shown in Figure 4.14—and see that it is transitively invoked from the `run()` method we have just finished annotating. This explains the presence of the `TabViewTableGen` color in the color environment. But why is the constraint for `DigitSequenceEditor.getFields()` " `GUI`"? We answer this question by considering the inheritance hierarchy. Class `DigitSequenceEditor` extends `KPanel` which in turn extends `JPanel`. Both of the latter classes are part of the Swing GUI framework. So the TimeField data really is part of the Swing GUI, and should only be accessed from the `AWT` thread. Thus, the color model for `DigitSequenceEditor.getFields()` requires that it be invoked only from the `AWT` thread. The call chain from the TabViewTableGen thread's `run()` method means that it is also invoked from that thread. This is a real error.

The consequence of this error takes two forms. First, there is a state consistency error on the time fields. This could cause the reading thread to see old or inconsistent data due to partial assignment. Worse, however, is the possibility of a safe publication error for the time fields. The Java memory model guarantees that any data modification made before a lock will be seen by any other thread that later acquires the same lock. This is "safe publication." Lack of safe publication can cause the reading thread to see data that is any of:

- Up to date
- Out of date
- Inconsistent
- Total garbage

This anti-pattern is widespread throughout SVC. Careful searching of the source code, however, does not reveal any errors of this kind involving fields that are declared directly as members of Jxxxx classes; the errors appear only for methods and fields that are declared as members of classes that extend Jxxx classes.

[6]Small print may make the text of the message difficult to read, but scaling issues prevent making the screen-shot larger. The three interesting lines say "Color model not consistent with code at call to org.shetline.skyviewcafe.DigitSequenceEditor.getFields(). at MainSkyViewPanel.java line 621", "1 precondition:", and "supporting information: Local color environment is (!java.awt.AWT & TabViewTableGen) | (java.awt.AWT & !java.awt.Compute) | (!java.awt.AWT & java.awt.Compute & TabViewTableGen); constraint is GUI".

Analysis I hypothesize that the author got the message about the change in thread usage policy in the Java 1.4 frameworks and carefully updated his old code to comply with the new policy. He appears to have accomplished this by using his IDE to search for field and method accesses to AWT and Swing fields and methods and carefully moved them to occur only on the `AWT` thread. I further hypothesize that either he was unaware that his own classes that extend AWT and Swing classes must follow the same rules, or that his IDE did not point out these methods and fields as part of his search. In either case, we see here that even a very careful programmer makes mistakes when trying to comply with thread usage policy. The thread coloring analysis, however, finds the problem.

The "IsEnabled()" pattern

Returning to Figure 4.12 we see a pair of interesting calls on lines 6 and 26. What's going on here? In this particular case, the implementer is disabling the "print" button at the beginning of the printing process, then enabling it once printing is complete. This prevents the user from requesting another printout while the application is busy producing this printout. The presence of these calls adds to our certainty that there will be only one `TabViewTableGen` thread at a time.

There is an underlying pattern here: "disable some portion of the GUI while my application is working with it, and enable that portion again when the application is done." The intent behind this pattern is to further enforce single-threaded behavior by preventing the user from making changes that would interfere with the running background activity. We'll see a broader use of this pattern in the JHotDraw case study in Section 4.6 on page 100.

4.5 Electric case study

In this case study I demonstrate that the thread coloring analysis easily scales to a 140KSLOC system, with techniques for near-linear-cost scaling to much larger systems. I also show analysis techniques used to reduce user annotation effort and some simple source code modifications that clarify design intent while further reducing user annotation effort.

The Electric case study is a hybrid of the laboratory and field case study types. I began with laboratory work on the source code, then visited the developers on-site for two and a half days, then followed up with several additional rounds of laboratory study. I undertook the study intending to demonstrate scalability in terms of code size. Electric provided additional challenges in the areas of model expressiveness and Developer ROI.

Thread usage patterns in this case study include the GUI pattern we have seen in previous case studies, single-writer multiple-reader access to a data repository, and GUI access to the data repository. Perhaps the most interesting result of this case study is the very low number of annotations per KSLOC needed to complete thread coloring analysis on Electric: currently 6.3 annotations per KSLOC, reduceable to 0.25 annotations per KSLOC with some additional engineering effort. See Section 3.5.2 on page 62 for detailed counts of annotations.

4.5.1 The application

Electric is an open-source VLSI-design tool, which was originally written in C, then translated into C++, then Java. It is nearly twenty years old. It is used by a wide variety of design teams and is actively maintained. Version 8.0.1 of Electric contains roughly 140KSLOC, spread over forty-four packages. One reason for the translation to Java was to take advantage of language support for multi-threading.

The intended structure of Electric as explained by the developers is as follows[7]:

> There is one persistent user-thread called the DatabaseThread, created in com.sun.electric.tool.Job. This thread acts as an in-order queue for Jobs, which are spawned in new threads. Jobs are mainly of two types: Examine and Change. These Jobs interact with the "database", which are objects in the package hierarchy com.sun.electric.database.
>
> The Rules are:

[7]Private communication.

```
1  /**
2   * @ColorImport java.awt
3   * @ColorDeclare DBChanger, DBExaminer
4   * @MaxColorCount DBChanger 1
5   * @IncompatibleColors DBChanger, DBExaminer, AWT
6   */
```

Figure 4.15: Thread coloring annotations for Electric

1. Jobs are spawned into new Threads in the order they are received.
2. Only one Change Job can run at a time.
3. Many Examine Jobs can run simultaneously.
4. Change and Examine Jobs cannot run at the same time.
5. Examine Jobs can only look at the database.
6. Change Jobs can look at and modify the database.

Because only one Change Job can run at a time, the Change Job is just run in the DatabaseThread. Examine Jobs are spawned into their own threads, which terminate when the Examine Job is done.

The Main thread merely sets up the GUI, runs a database initialization Job, and then terminates.

Armed with this explanation and our understanding of the basic Swing thread usage policy, we can begin to analyze Electric. Our first goal is to assure that the various Jobs actually obey the reader-writer discipline as specified.

4.5.2 Case study details

Thread colors

Because Electric has a Swing-based GUI, we begin with the standard Swing thread colors: `AWT` and `Compute`. In addition, we need to distinguish threads that may both read and write the database from threads that should only read from the database. To this end we declare two new thread colors as shown in Figure 4.15. We intend that the `DBChanger` color be used for all Jobs that may modify the database. The `DBExaminer` color should be used for all Jobs that may only read the database. The `@ColorImport` annotation on line 2 gives us visibility to the standard `AWT` and `Swing` thread color definitions. On line 3 we declare our new thread colors. Line 4 notes that there is at most one `DBChanger` thread at a time, while line 5 declares that no thread may simultaneously be associated with the `DBChanger`, `DBExaminer`, or `AWT` colors. This property allows us to conclude that a thread that has one of these colors certainly does *not* have either of the others; that is, threads with these colors are distinct at any given point in time. Note that if we had followed only the developer's explanation and ignored the Swing GUI, our annotation might have read "`@IncompatibleColors DBChanger, DBExaminer`." This would have been wrong because it would have ignored the existence of the AWT event thread.

Analysis This is the first case study presented where we have design intent expressed directly by the developers. Both of the studies presented previously required substantial investment of time and energy to reverse-engineer the intended thread usage patterns in the software. That was not an issue here. Instead, I was able to concentrate on expressing the design intent and on ways to reduce the effort required for that expression.

The Electric developers were able to provide a thread usage policy, in part, because they had only recently been through the effort of developing one. The decision to translate Electric from C++ to Java was largely motivated by a desire to take advantage of Java's support for concurrency—the C++ version of Electric was entirely single-threaded. Thus, concurrency and safe shared access to the database were very much part of the Electric developers' daily work at this particular point in time. They were further motivated by the fact that their user base was complaining about frequent "mysterious crashes" and "random exceptions from inside the tool." The Electric development team immediately focused on the newly-added concurrency as the likely culprit.

Annotating class `Job`

Now that we have defined the colors we intend to use, we must decide where to grant those colors. We want to "stamp" threads with their colors at program points where we can be certain what role is intended for the current thread. Based on the explanation from the developers, we hypothesize that the creation points of jobs are what we're looking for.[8] We'll check this by examining the source code.

Figures 4.16 on the next page and 4.17 on page 94 show the relevant code extracted from **`class`** `Job`. On line 13 we see the declaration `DatabaseChangesThread`, the one and only thread that is allowed to change the database. Just above that, on line 10 we encounter our first sign of trouble. There are actually *three* kinds of jobs, not two. After brief consideration, we decide that `UNDO` jobs should be modeled as `DBChanger` (just like `CHANGE` jobs).

On line 17 is the `run()` method for `DatabaseChangesThread`. Normal thread usage patterns would indicate that this method is executed as the 'main body' of its thread. Accordingly, we annotate it with a `@Grant DBChanger` annotation. And, sure enough, there is an invocation of the `run()` method of some `Job` at line 20.

Moving on to method `waitChangeJob`, we see that `EXAMINE` jobs are handled separately from `CHANGE` and `UNDO` jobs; this raises our confidence in our choice of thread colors for this modeling effort. There are three other items of particular interest in this part of Figure 4.16:

- On line 32 we see code to track the number of outstanding `DBExaminer` jobs. There is a matching decrement on line 54.
- On line 34 we see that each `DBExaminer` is started in a new thread of its own. This is consistent with the threading model described by the developers.
- On line 39 we see that `CHANGE` and `UNDO` jobs are permitted to continue only when there are no outstanding `EXAMINE` jobs. This too is consistent with the threading model described by the developers.

Figure 4.17 on page 94 shows additional code from **`class`** `Job`. We're interested in the `run()` method. The first thing we observe is that it handles `CHANGE`, `EXAMINE`, and `UNDO` jobs separately. This makes it easy for us to tell where to note our knowledge of thread colors.

We next consider our knowledge of this `run()` method's call stack. It will be invoked either from the `DatabaseChangesThread`'s `run()` method (see line 20 of Figure 4.16 on the facing page) or from the Java run-times after an `EXAMINE` thread is started on line 34 of Figure 4.16. For purposes of our thread usage model, this means that it could be called from either a `DBChanger` thread or a `DBExaminer` thread. The `@Color` annotation on line 62 says exactly this.

Although we have constrained the `run()` method to be called only from `DBChanger` and `DBExaminer` threads, we have not yet distinguished which of those alternatives actually made the call. At lines 66, 71, and 75 the developers have written code that clearly distinguishes these cases for us. The pairs of `@Revoke` and `@Grant` annotations at these lines manipulate the current color environment so that we are sure that it is `(DBChanger & !DBExaminer)` at lines 66 and 75 and is `(DBExaminer & !DBChanger)` at line 71. Similarly, at lines 81 and 84 we also know exactly what the color environment should be and annotate accordingly.

Note that the code in the **`finally`** block (see line 80) is cleaning up from each kind of job by notifying the locking scheme that the job has completed. On the basis of examining and annotating **`class`** `Job`, we tentatively conclude that the developers indeed issue jobs according to a single-writer/multiple-reader discipline.

```
...
/**
 * @ColorDeclare DBChanger, DBExaminer
 * @IncompatibleColors DBChanger, DBExaminer, AWT */
public abstract class Job implements ActionListener, Runnable {
   /** Type is a typesafe enum class that describes the type of job */
   public static class Type {
      ...
      public static final Type CHANGE  = new Type("change");
      public static final Type UNDO    = new Type("undo");
      public static final Type EXAMINE = new Type("examine"); }
      /** Thread which executes all database change Jobs. */
      private static class DatabaseChangesThread extends Thread {
         ...
         private static int numExamine = 0;
         ...
         public void run() {//@Grant DBChanger
            for (;;) {
               Job job = waitChangeJob();
               job.run();
            }
         }
      }

      private synchronized Job waitChangeJob() {
         for (;;) {
            if (...) {
               Job job = ...;
               ...
               if (job.jobType == Type.EXAMINE) {
                  job.started = true;
                  numExamine++;
                  ...
                  Thread t = new Thread(job, job.jobName);
                  t.start();
                  continue;
               }
               // job.jobType == Type.CHANGE || jobType == Type.UNDO
               if (numExamine == 0) {
                  ...
                  return job;
               }
            }
            try { wait(); } catch (InterruptedException e) {}
         }
      }
      /** Add job to list of jobs */
      private synchronized void addJob(Job j) { ... }
      /** Remove job from list of jobs */
      private synchronized void removeJob(Job j) { ... }

      /** @ColorConstraint DBExaminer */
      private synchronized void endExamine(Job j) {
         numExamine--;
         if (numExamine == 0) notify();
      }
   }
   ...
```

Figure 4.16: Class `Job` from Electric (Part 1)

```
    ...
    //--------------------------PUBLIC JOB METHODS--------------------------
    /** Run gets called after the calling thread calls our start method
      * @Color (DBChanger | DBExaminer);
      */
    public void run() {
       try {
          if (jobType == Type.CHANGE) {//@Revoke DBExaminer
             //@Grant DBChanger
             Undo.startChanges(...);
             doIt();
             Undo.endChanges();
          } else if (jobType == Type.EXAMINE) {//@Revoke DBChanger
             //@Grant DBExaminer
             numExamine++;
             doIt();
          } else if (jobType == Type.UNDO) {//@Revoke DBExaminer
             //@Grant DBChanger
             changingJob = this;
             doIt();
          }
       }catch (Throwable e) { ... } finally {
          if (jobType == Type.EXAMINE) {//@Revoke DBChanger
             //@Grant DBExaminer
             databaseChangesThread.endExamine(this);
          } else {//@Revoke DBExaminer
             //@Grant DBChanger
             changingJob = null;
             ...
          }
       }
    }
 }
```

Figure 4.17: Class `Job` from Electric (Part 2)

```
private static class CheckSchematicHierarchically extends Job {
 ...
 protected CheckSchematicHierarchically(...) {
    super(..., Job.Type.CHANGE, ...);
    ...
    startJob();
 }
 public boolean doIt() {
    //@Grant DBChanger
   ...
   Schematic.doCheck(cell);
   ...
   return true;
 }
}
```

Figure 4.18: Example `CHANGE` `Job` from Electric 8.0.1

```
private static class CheckLayoutIncrementally extends Job {
 ...
 protected CheckLayoutIncrementally(...) {
    super(..., Job.Type.EXAMINE, ...);
    ...
    startJob();
 }
 public boolean doIt() {
     //@Grant DBExaminer
    ...
    int errorsFound = Quick.checkDesignRules(...);
    ...
    doIncrementalDRCTask();
    return true;
 }
}
```

Figure 4.19: Example `EXAMINE` `Job` from Electric 8.0.1

```
package com.sun.electric.database.network;
...
public class JNetwork {
 ...
 public JNetwork(Netlist netlist, int netIndex) {...}
 public void addName(String nm, boolean exported) {...}
 public Cell getParent() { ... }
 public int getNetIndex() { ... }
 public Iterator getNames() { ... }
 public Iterator getExportedNames() { ... }
 public boolean hasNames() { ... }
 public boolean hasName(String nm) { ... }
 public Iterator getPorts() { ... }
 public Iterator getExports() { ... }
 public Iterator getArcs() { ... }
 public String describe() { ... }
 public String toString() { ... }
 ...
}
```

Figure 4.20: A typical database class from Electric 8.0.1

Annotating individual `Job`s

We now investigate the creation points of jobs. Each class that defines a job contains a nested class along the lines of Figure 4.18 on the preceding page or Figure 4.19 on the facing page. Note that on line 4 of Figure 4.18 the developers have noted that `CheckSchematicHierarchically` is a `CHANGE` job (and similarly for the `EXAMINE` job in Figure 4.19). This pattern is found pervasively throughout Electric.

We annotate the `doIt()` method of each Job with a `@Grant` annotation that assigns the appropriate color for the job, as seen on line 9 of Figure 4.18 and line 9 of Figure 4.19. There are 155 Jobs in Electric 8.01, so we used 155 annotations to grant the correct thread color for each.

As a result of adding these `@Grant` annotations, the thread coloring analysis will propagate the relevant color environments around Electric's call graph using its inference algorithm. The color environment of all methods reachable from any `Job` will thus include `DBChanger`, `DBExaminer`, or both, as appropriate.

Annotating the database

Now that we have identified the points of creation of every `EXAMINE` and `CHANGE` job, and have annotated them to grant appropriate thread colors, we'd like to check the usage of the accessor methods in the database. Specifically, we want to assure that only `CHANGE` jobs may modify the database, and that only `EXAMINE` or `CHANGE` jobs may read the database.

Figure 4.20 shows extracts of the code from `JNetwork.java`, a typical class from the database. We see accessor methods with names starting with "get" and "has"; we infer that such methods only `EXAMINE` the database. We also see methods with names starting with "add"; we infer that such methods `CHANGE` the database.

We'll annotate these methods with color constraints that require color `DBChanger` for the `CHANGE` methods, and that require `(DBChanger | DBExaminer)` for the `EXAMINE` methods. Because there are thousands of accessor methods, we chose not to write individual annotations. Once again, we fall back on Halloran's scoped promises.

The scoped promises used in this case study take the form "`@Promise "payload_annotation" for location_specifier`" where `location_specifier` is a modified regular expression that is matched against methods in the scope containing the `@Promise`. The individual terms of the `location_specifier` are method names that are wild-carded more-or-less as Unix shell expressions; arguments are specified positionally, with normal wildcarding. The special argument wild-card "**" matches any number of arguments of any type. `Location_specifier` terms are combined with and ("&"), or

[8] In this discussion I use `Job` or **`class`** `Job` to refer to the class named "Job," and use "Job" or "Jobs" to refer to instances of that class. The notional "jobs"—that is, the activities that take place to change or examine the database—are referred to as "job" or "jobs," sometimes with the name of a `Type` or a thread color (*e.g.* `DBChanger`) as a modifier.

```
/** @Promise "@Color (DBExaminer | DBChanger)" for get*(**)
  * @Promise "@Color (DBExaminer | DBChanger)" for  is*(**) | same*(**)
  * @Promise "@Color (DBExaminer | DBChanger)" for  compare(**) | connectsTo(**) | contains*(**)
  * @Promise "@Color (DBExaminer | DBChanger)" for describe() | find*(**) | num*(**)
  * @Promise "@Color DBChanger" for set*(**) | make*(**) | modify*(**) | clear*() | new(**) | add*(**)
  */
package com.sun.electric.database.network;
```

Figure 4.21: A typical `package-info.java` file from Electric's database

("|"), and not ("!") operators and parentheses. Thus, the `location_specifier` "`set*(**)| make *(**)`" matches any method in scope whose name begins with either "set" or "make" and with any number of parameters of any type.

Figure 4.21 shows an extract from a typical `package-info.java` file from the Electric database. On lines 1 to 4 we see `@Promise` annotations that mark getters in this package as usable by both `DBChanger` and `DBExaminer` threads. On line 5 we see a `@Promise` annotation that marks setters as usable only by the `DBChanger` thread.

Annotating the GUI

Electric's user interface is a standard Swing UI. As a result, nearly all annotation for the UI code is inherited from the Swing libraries and need not be written by the user.[9] As expected, the vast majority of methods inherit the color constraint `@Color AWT`, which indicates that they are to be invoked only from the AWT/Swing event thread. The only user-written annotation needed in the Electric GUI code was to mark a few `run()` methods as being `@Color AWT`. These methods were all being used as `Runnables` passed to `SwingUtilities.invokeLater()` in accordance with normal Swing practice.

Analysis

Electric 8.0.1 is a medium-sized program of about 140KSLOC. This is large enough to serve as an interesting test of whether the thread coloring analysis and prototype assurance tool can analyze substantial programs at reasonable cost to the user (in terms of time and effort). In Section 3.5.2 on page 66 we described the use of modules to divide Electric into separately analyzed parts. In this section we examine other techniques to facilitate static analysis at scale: scoped promises, inheritance of coloring annotations, and annotation inference.

Scoped promises This technique, due to Halloran [25], takes a payload annotation and uses wildcarding over the names of Java entities to identify the specific Java entities to which it should be applied. The wildcarding style is patterned after that used in the AspectJ language [60]. Scoped promises are "scoped" in that they apply to any matching Java entity located within the same static scope as the scoped promise.

The primary use of scoped promises in Electric is to reduce the number of user-written annotations needed for analysis, and so reduce the user's cost of analysis. For example, the database contains 1731 Java entities that are referenced from other modules. Because of their cross-module references they must each be annotated with thread coloring constraints. With scoped promises, we were able to avoid writing nearly two thousand annotations, and instead write only fifty-four—six scoped promises in each of nine packages. Figure 4.21 shows samples of these promises. This number could be reduced to six by upgrading the Fluid system's support for scoped promises to include a module scope and then replacing the package-scoped promises with module-scoped promises.

This technique is effective because programmers attempt to choose meaningful names for methods as an aid to comprehension. These names are often highly stylized. Indeed, as shown above, we see this style of naming in Electric's database. Wildcarding of names would provide less benefit if programmers chose names in a more random fashion.

[9] Actually, when doing the case study I had to write the needed annotations. They are, however, notionally part of the library and would be provided as part of a "real" assurance tool.

```
 1  public abstract class ExamineJob extends Job {
 2    ...
 3   protected ExamineJob(...) {
 4       super(... , Job.Type.EXAMINE, ...);
 5       ...
 6       startJob();
 7   }
 8   /**
 9    * @Color DBExaminer
10    */
11   public abstract boolean doIt();
12
13  //——————————————PUBLIC JOB METHODS——————————————
14      /** Run gets called after the calling thread calls our start method
15        * @Color DBExaminer
16        */
17      public void run() {
18          try {
19                numExamine++;
20                doIt();
21          }catch (Throwable e) { ... } finally {
22                databaseChangesThread.endExamine(this);
23          }
24      }
25  }
```

Figure 4.22: New `ExamineJob` class for Electric 8.0.1

Inheritance of thread coloring annotations Methods that override methods from parent classes inherit the thread coloring annotations found on the overridden method by default. This inheritance is transitive; the parent method may well have inherited the annotation from its parent, and so on. This default behavior also holds for implementation of Java `interface` methods. I chose this default in order to reduce the number of annotations written by users.

For example, Electric's user interface is built on the Swing framework. Due to default inheritance, however, the majority of methods in Electric's UI needed no user-written annotation; they inherited the necessary annotations from the framework code. This default inheritance is not a small matter for user interface code. Of the 3599 methods in Electric's UI, roughly 2700 (or about 75%) inherit their color constraints from the Swing framework with no user action required.

Annotation inference We use inference to discover the color constraints of methods that are not visible across module boundaries. This technique reduces the number of annotations that a user must write in order to perform complete thread coloring analysis of his application. How much does this reduce the number of methods requiring annotation?

There are nine thousand eight hundred and fourteen (9814) methods and constructors in Electric v8.0.1. Of these, approximately 80% are confined to a single module; nearly all of these have their color constraints inferred during analysis. Inferring constraints for non-API methods reduces the number of methods and constructors requiring some form of annotation by nearly 4x.

Changing source code

Up to this point, none of the techniques used to assist with issues of scale required modification of Electric's source code. By making a small modification to Electric's class hierarchy, we can eliminate another one hundred and fifty-two annotations. We accomplish this by introducing three new classes, each of which extends class `Job`. We'll create one class each for `EXAMINE`, `CHANGE`, and `UNDO` jobs. Figure 4.22 shows the text of our new `ExamineJob` class. The key change is that we constrain the `doIt()` method of `ExamineJob` to be invoked only by the `DBChanger` or `DBExaminer` threads (as seen on line 15), and constrain the `doIt()` methods of `ChangeJob` and `UndoJob` to be invoked only from the `DBChanger` thread. We then modify each creation point for a job to use the appropriate newly written subclass of Job. Because we propagate annotations from parent classes to child classes, the individual Job creation sites no longer require annotation to indicate the desired thread usage; they inherit the constraint from their parent class instead.

```
 1  package com.sun.electric.tool.user;
 2  /* Class for highlighting of objects on the display. */
 3  public class Highlight {
 4      ...
 5      /** The highlighted polygon */
 6    private Poly polygon;
 7      ...
 8      /** Method to display this Highlight in a window.
 9       * @Color AWT
10       */
11      public void showHighlight(...)      {
12          if (...) {
13              ...
14              // draw outline of poly
15              ... = (polygon.getStyle() == ...);
16              drawOutlineFromPoints(..., polygon.getPoints(), ...);
17              ...
18              return;
19          }
20          ...
21      }
22      ...
23  }
```

Figure 4.23: Extracts of class `Highlight` from Electric 8.0.1

```
package com.sun.electric.database.geometry;
public class Poly implements Shape{
    /** @Color (DBChanger | DBExaminer); */
    public Poly.Type getStyle() { return ...; }
    ...
    /** @Color (DBChanger | DBExaminer); */
    public Point2D [] getPoints() { return ...; }
    ...
}
```

Figure 4.24: Extracts of class Poly from Electric 8.0.1

In addition, the `run()` method for each subclass can be substantially simplified. At line 15 of Figure 4.22 on the previous page we see the `ExamineJob` version of this method. Note that it is declared to be a `DBExaminer`-colored method, and that its body has been simplified to contain only the portions of the original `run()` method that pertain to `EXAMINE` jobs. Similar simplifications apply to `CHANGE` and `UNDO` jobs as well.

Attentive readers will have noted that the instances of `ExamineJob`, `ChangeJob` and `UndoJob` are not fully substitutable for all instances of their parent class. This may appear unfortunate but it is not, in fact, a newly introduced issue in the code. Prior to our introduction of these new subclasses, individual instances of **`class`** `Job` already suffered from this problem; some due to the difference between `EXAMINE`, `CHANGE`, and `UNDO` Jobs, and others because they are actually implementing different operations. In either case, we have not introduced any new incompatibility.

Initial consistency check

On running the thread coloring analysis, we discover inconsistencies between Electric's as-written code and the model we have expressed.

Figure 4.23 shows extracts from class `Highlight`, one of the problematic GUI classes. The tool discovers by inference on the call graph that the `showHighlight` method on line 11 is invoked only from the `AWT` thread; its color constraint is thus inferred to be `@Color AWT`. But on line 15 it calls `polygon.getStyle()` and on line 16 it calls `polygon.getPoints`, both of which are part of the database API. Figure 4.24 shows the relevant code, with in line annotations that match the effect of the scoped promises we actually used. Clearly we are calling these "`@Color (DBChanger | DBExaminer)`"

```
1  if (Job.acquireExamineLock(block)) {
2  // block true==>wait, false==>fail fast
3       try {//@Grant DBExaminer
4           // do database Actions
5       } finally {
6           Job.releaseExamineLock();     // release lock whether exception or no
7       }
8  } else {
9      // put up the 'busy' cursor
10     ...
11     // then...
12      Job.invokeExamineLater(Job, Key); //Job is the database actions not done above
13  }
```

Figure 4.25: Typical `ExamineLock` usage from Electric 8.0.1

methods from the wrong thread! Specifically, the GUI calls "`get*(**)`" methods on database objects without participating in the reader/writer locking scheme implemented in class `Job`. This does not match the explanation given by the developers.

Examination of other findings of thread coloring inconsistency reported by the tool show that bugs of this sort are extremely common throughout the GUI code. Interestingly, the `Jobs` all turned out to obey their intended roles: all the `EXAMINE` jobs used read-only methods and avoided writing to the database.

A query to Electric's developers yielded the answer. It seems that as newcomers to Java programming they had forgotten that the GUI always executes from the `AWT` event thread. As a result, the GUI code freely read the database from the `AWT` event thread without participating in the reader/writer locking scheme used by all Jobs. This bug caused a variety of problems including both concurrent modification exceptions from their own code and internal exceptions originating deep inside the Swing framework. Electric's developers had not identified the issue until after they sent the description of their intended threading rules (quoted above). They did, however, identify it before my query about the issue. The Electric development team invested "weeks of time" from "multiple developers" to chase down the source of their "random crashes." Finding the problem independently with the Fluid prototype assurance tool and thread coloring took about eight working hours spread over several days.

Fixing the GUI thread usage bug in version 8.0.1

The Electric developers made two efforts to correct their thread usage policy and so obey both the Swing thread usage policy and their intended reader-writer policy with respect to the database. Their first attempt was to define a new kind of `Job` that would provide its own thread for execution—
the `InThreadExamineJob`. Their intent was that this would be used only for the GUI's event thread. An `InThreadExamineJob` runs immediately if no `CHANGE` jobs are in progress or queued, otherwise it is queued for later execution. To support this model they also added a new "`ExamineLock`," used to indicate that one or more `EXAMINE` jobs are currently executing. This model allows GUI methods to access the database while still respecting the reader-writer discipline needed for correctness. The cost of this model is that GUI updates could be substantially delayed by long-running `CHANGE` jobs.

Figure 4.25 shows a typical use of the `ExamineLock`. The code shown would be executed directly from the GUI event thread. Modeling the intended thread usage policy is straight-forward. At line 3 we add an annotation to grant the `DBExaminer` color. This is safe because we are inside the acquisition of the `ExamineLock`. As usual, the granted color will expire at the end of the lexical block that touches the database, so the color will not propagate outwards onto arbitrary GUI code.

This model works, in the sense that GUI actions that conform to the thread usage policy can safely access the database. Examination of the Electric change logs shows, however, that there were repeated bugs due both to direct calls from the GUI thread to database methods without using this `ExamineLock` discipline and also due to direct invocation of `CHANGE` Jobs from the GUI thread. This latter group of problems would violate the assumptions of the `ExamineLock` discipline even when the developers remembered to use it.

Analysis I hypothesize that these bugs could be quickly discovered with the prototype assurance tool—this was certainly true of the few I examined directly. If so, integration of such a tool would have allowed the

Electric developers both to reduce their debugging and maintenance costs and to stick with this version of their thread usage policy. Lacking such a tool, and suffering from repeated failures due to re-introduction of "wrong thread" bugs, they decided to try again.

Fixing the GUI thread usage bug in version 8.0.5

In version 8.0.5 the Electric developers changed their thread usage policy again. The new policy splits the application into a GUI client application and a separate headless server application. The two applications may be run on the same machine; alternatively, the server can be run on a remote machine.

The client application views the database as a read-only resource. Clients can choose to run `EXAMINE` jobs either locally or remotely. All changes to the database are sent to the remote server as `CHANGE` Job requests. The remote server responds either with a set of deltas to the client's existing database or with an entire new database. Database deltas are read-only objects and do not cause modifications to the client's read-only copy of the database. Instead the client prefers data from the newest possible delta, then the next older, and so on in reverse time order, finally falling back to the database if no data was previously found.

This organization allows the client's GUI to have full access to the database and deltas at any time. Further, the read-only nature of this data avoids the need for locking. The only portion of the client that requires any locking whatsoever is the reverse-time-order list of deltas due to the potential for updates from the server. Meanwhile, the server implementation is free to follow the original reader-writer locking scheme without the complication of any threads beyond those associated with `Jobs`.

We have not yet modelled the thread usage policies in this version of Electric. A brief initial investigation suggests that a thread usage model for version 8.0.5 will be no more complex than those for prior versions.

4.6 JHotDraw case study

JHotDraw is another AWT/Swing GUI application. In this case study I examine its extension of the standard AWT/Swing thread usage policy, used to support per-project background work threads, along with the life-cycle of those threads. I also show how to use thread coloring to ensure that background threads are started only when their project is "disabled," and find that JHotDraw's thread usage policy appears to be in conflict with the AWT/Swing thread usage policy.

This is a laboratory-only case study, performed without access to the developers or to any documents other than those distributed with the source code. This lack of access limited my ability to perform useful analysis. Accordingly, this case study ends with observations on assurance possibilities that might have been addressed if additional thread usage policy intent had been available.

I have also omitted an additional case study performed on the open-source editing program JEdit, as the results were essentially similar to those reported here for JHotDraw.

4.6.1 The application

JHotDraw is an open-source graphic editor that supports operation on many file formats. JHotDraw makes extensive use of design patterns after the fashion of the Gang-of-four book [22]; it contains fairly extensive documentation of the patterns used and of which classes/subsystems embody them. JHotDraw is a moderate-sized application, with around 400 classes and a few-thousand methods and constructors; see Table 4.3 for details. JHotDraw follows the basic multi-threading mandated by the AWT/Swing thread usage policy; it is not as heavily multi-threaded as Sky View Café, for example.

4.6.2 Case study details

In order to gain any benefit from thread coloring, we need to start by understanding the thread usage policies in JHotDraw. Lacking direct developer input, we'll have to rely on searching through the code and documentation along with our reverse engineering skills. As usual, we use Eclipse's searching tools to find `Runnables` and `Threads`, classes that extend them, classes that import anything from `java.util.concurrent`, and `run()` and `start()` methods of the classes thus identified as being interesting; we also examine the documentation that is included in the JHotDraw source distribution.

Packages	Classes	Methods	Constructors	Synch Methods	Synch Blocks	Per-Project Worker Threads Started	Other Worker Threads Started
42	396	3818	456	6	4	10	8

Table 4.3: Raw counts of entities in JHotDraw

```
 1  /**
 2   * Executes the specified runnable on the worker thread of the project.
 3   * Execution is perfomred sequentially in the same sequence as the
 4   * runnables have been passed to this method.
 5   */
 6  public void execute(Runnable worker) {
 7     if (executor == null) {
 8        executor = Executors.newSingleThreadExecutor();
 9     }
10     executor.execute(worker);
11  }
```

Figure 4.26: `Execute` method from JHotDraw's `Project` class

The documentation tells us that JHotDraw's unit of work is the `Project`; by default it creates a `Project` for each file the user opens for viewing or editing. Each project has a main window in which it is displayed. In accordance with the AWT/Swing thread usage policy, JHotDraw mandates that long-running actions should be run on a background thread rather than on the `AWT` event thread.

By searching we find that, as one might expect for a basic drawing application, the bulk of JHotDraw's code is its user interface, nearly all of which executes on the `AWT` thread.

`Projects` and `Executors`

We begin by looking at the `Project` class, portions of which are shown in Figure 4.26. There we see that each `Project` has an `execute(...)` method and a private `executor` (seen on line 21 of Figure 4.27 on the following page). The `execute(...)` method seen in Figure 4.26 always hands its `Runnable` argument to the private `executor` for execution. Because the `executor` is initialized as a `SingleThreadedExecutor` (as seen on line 10), we know both that each project has its own `executor` , and that there is no more than one such `executor` per project. These properties allow us to conclude that background jobs started via this interface will be executed sequentially for any given project.

Next, we glance through the code in the rest of class `Project`. On lines 7 through 18 of Figure 4.27 we see a key comment describing the intended mechanism used to prevent background jobs from executing in parallel. Although the documentation is quite extensive in terms of design patterns and their realization, the thread usage policy is not described therein—this comment is the only mention of the policy that I have found in JHotDraw. There is no enforcement of this thread usage policy anywhere in JHotDraw.[10]

This usage of the "enabled" property appears to serve three different purposes. The first, which is directly mentioned in the comments, is to prevent multiple sequential jobs from running simultaneously. The second purpose is to inform the GUI that the project has been disabled, which should cause the GUI to reject UI actions on the project. This serves to prevent the user from making changes while a sequential job is running. The third purpose is to allow the sequential job to touch the parts of the project's GUI from a non-`AWT` thread. As we shall see, this third usage is problematic.

`Workers`

When searching for `Runnables` and `invokeLater()` calls we find a helper class `Worker`, seen in Figure 4.28 on the next page, that provides a common model for tasks to be run on a `Project`'s background

[10]This policy appears to be a perfect use-case for Fairbanks' "Design Fragments" [14, 13] in combination with thread coloring.

```
/** Returns the enabled state of the project. */
public boolean isEnabled();

/**
 * Sets the enabled state of the project.
 *
 * The enabled state is used to prevent parallel invocation of actions
 * on the project. If an action consists of a sequential part and a
 * concurrent part, it must disable the project only for the sequential
 * part.
 *
 * Actions that act on the project must check in their actionPerformed
 * method whether the project is enabled.
 * If the project is disabled, they must do nothing.
 * If the project is enabled, they must disable the project,
 * perform the action and then enable the project again.
 *
 * This is a bound property. */
public void setEnabled(boolean newValue);
...
final private java.util.concurrent.Executors executor = Executors.
      newSingleThreadedExecutor();
```

Figure 4.27: Code and comments from JHotDraw's `Project` class

thread. The `construct()` method provided by the user is run on whatever thread eventually executes the `Worker`'s `run()` method (see line 14), after which the user's `finished()` method is invoked on the `AWT` thread (line 18).

```
1  public abstract class Worker  implements Runnable() {
2   private Object value;
3   public abstract Object construct();
4   public abstract void finished(Object value);
5   ...
6   public final void run() {
7      final Runnable doFinished = new Runnable() {
8         /** @ColorConstraint AWT; */
9         public void run() {
10           finished(getValue());
11        }
12     };
13     try {
14        setValue(construct());
15     } catch (Throwable e) {
16        e.printStackTrace();
17     } finally {
18        SwingUtilities.invokeLater(doFinished);
19     }
20  }
21  ...
22  public void start() {
23     new Thread(this).start();
24  }
25 }
```

Figure 4.28: Class `Worker` from JHotDraw

Typical `Worker` usage Now that we've found the `Worker` class and have determined that it is interesting, we need to consider how it is used. Looking through the source code for uses of `Workers`, we see that `Workers` in JHotDraw are used in two different ways. The first pattern, as seen in Figure 21, is to use the `Worker` as the `Runnable` that executes the sequential work for a `Project`. In this pattern the programmer disables the `Project` (line 3), then executes the **`new`** `Worker()`, thus running it on the `Project`'s background thread. Note that the matching call to re-enable the `Project` is placed in the `finished(...)`

method (line 17) which is guaranteed to run on the `AWT` event thread—a good thing because the implementation of `Project.setEnabled(`**`boolean`**`)` is eventually provided by `java.awt.Component,` whose `setEnabled` method may be called only from the `AWT` event thread. The `Worker`'s `construct()` method holds the code that performs lengthy sequential actions – file I/O in this case. Figure 4.29 shows the life-cycle of this `Worker` pattern.

The other `Worker` usage pattern we find is used for background jobs for which there is no `Project`. Examples include loading Applet preferences during initialization. In this usage—seen in Figure 17—the `Worker`'s `construct()` method once again kicks off the action. This time, however, the `Worker` may or may not be started from the thread, it generally has an empty `finished()` method, and creates and then is run on its own background thread via the start call at line 16. Figure 4.30 shows the life-cycle of this `Worker` usage.

Actions

Remember the comment we saw back in class `Project`? It tells us about rules we must follow when performing actions (line 7 of Figure 4.27 on the facing page). We'd better find those actions, and see what they really do. When we go look at the actions, we find that JHotDraw has ten `Workers` that are executed in the `Project`'s background thread, seven `Workers` that execute in their own background thread, and one `Worker` that executes in the `AWT` event thread via `invokeLater()`.

The comments we have seen previously in Figure 4.27 tell us that some actions must be executed sequentially—these appear to be actions that modify project-wide data—while others may execute in parallel. I was unable to find either comments or documentation that indicates which actions are expected to fall in which category. I did observe, however, that actions such as reading and writing files, and creating new editors use the sequential `Worker` pattern. Simple drawing actions do not use a `Worker` at all, but rather execute directly in the `AWT` event thread. I did not see a unifying theme to the actions that use the parallel `Worker` pattern.

Coloring `Workers`

The choice of coloring strategies for JHotDraw's `Workers` depends on our goal: what properties do we wish to enforce? One reasonable choice would be to require that `Project Workers` execute only when the `Project` is not enabled. To this end we declare a thread color `ProjDisabled` that we intend to grant only when a `Project` has been disabled. We will use this color essentially as a global flag. We also declare colors `ProjWorker` and `FreeWorker`. We'll grant `ProjWorker` when we are executing from the `construct()` method of a `Project`'s `Worker`; we'll grant `FreeWorker` when executing from the `construct()` method of a non-`Project Worker`. Figure 4.31 on page 105 shows the color declarations and their associated incompatibilities. We declare that the two `Worker` colors are incompatible with the `AWT` color to indicate that they are known to be granted for threads that are never the `AWT` thread. We omit `ProjDisabled` from this declaration, because the enabled status of a `Project` is quite independent of which thread is currently executing.

To use our newly declared colors we introduce two new subclasses of `Worker` (seen in Figure 4.32 on page 107), one for each of the two varieties described above. There are two reasons for the introduction of these subclasses. First, as in Electric with subclasses of `Job`, we have added thread coloring annotations to each of these subclasses so that users of these classes inherit those annotations, and need not write additional annotations of their own. Second, we partially work around a limitation of the current prototype assurance tool: it has no support for parameterized thread colors, which we would need to fully stitch together the `Runnables`, their thread colors and the various threads (the `executors` from the `Projects` and the new thread in the `Worker`'s `start()` method).

To help programmers in choosing the correct sub-class of `Worker`, we modify the code of the `execute(...)` method for all `Projects` to take a `ProjectWorker` as its argument instead of accepting any `Runnable` as before (see Listing 4.3). The change of argument type forces programmers to choose a `ProjectWorker` for all of their project-specific background jobs. The `@Color` annotation requires that the `execute(...)` method be invoked only when we know that the `Project` is disabled. The coloring annotations seen in Figure 4.32 on page 107 lines 2 to 18 note that the `construct()` method will execute on

```
/** running in the AWT thread... */
 ...
 project.setEnabled(false);
 // Open the file
 project.execute(new Worker() {
          public Object construct() {
    try {
        project.read(file);

        return null;
    } catch (IOException e) {
        return e;
    }
 }
 public void finished(Object value) {
    ...
    setEnabled(true);
    ...
 }
});
```

Listing 4.1: JHotDraw typical sequential `Worker` usage

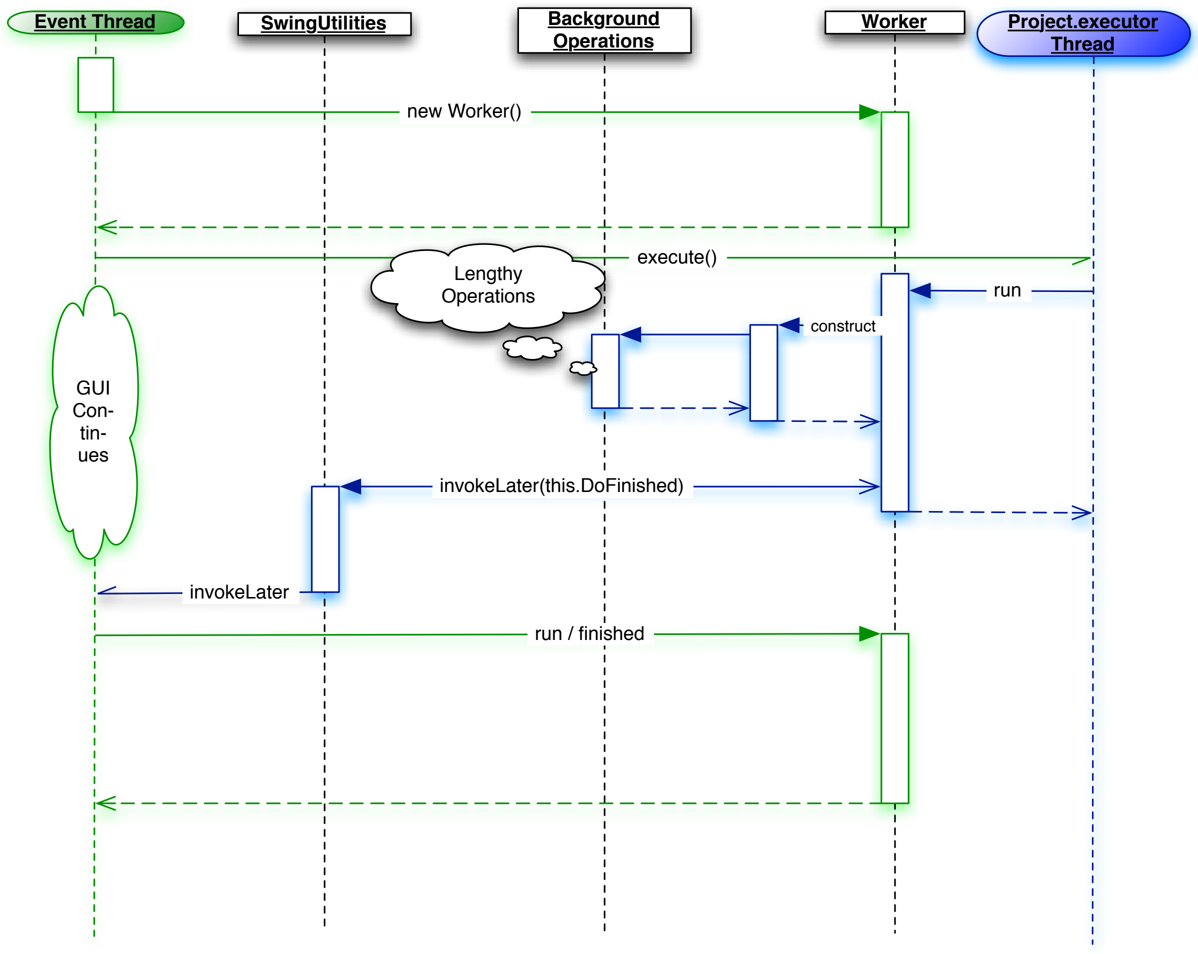


Figure 4.29: JHotDraw sequential `Worker` life-cycle

```
/** running in the AWT thread... */
 ...
 // Open the file
 new Worker() {
            public Object construct() {
      try {
          // do something slow that can run in parallel
          return null;
      } catch (IOException e) {
          return e;
      }
 }
 public void finished(Object value) {
     null;
 }
}.start();
```

Listing 4.2: JHotDraw typical parallel `Worker` usage

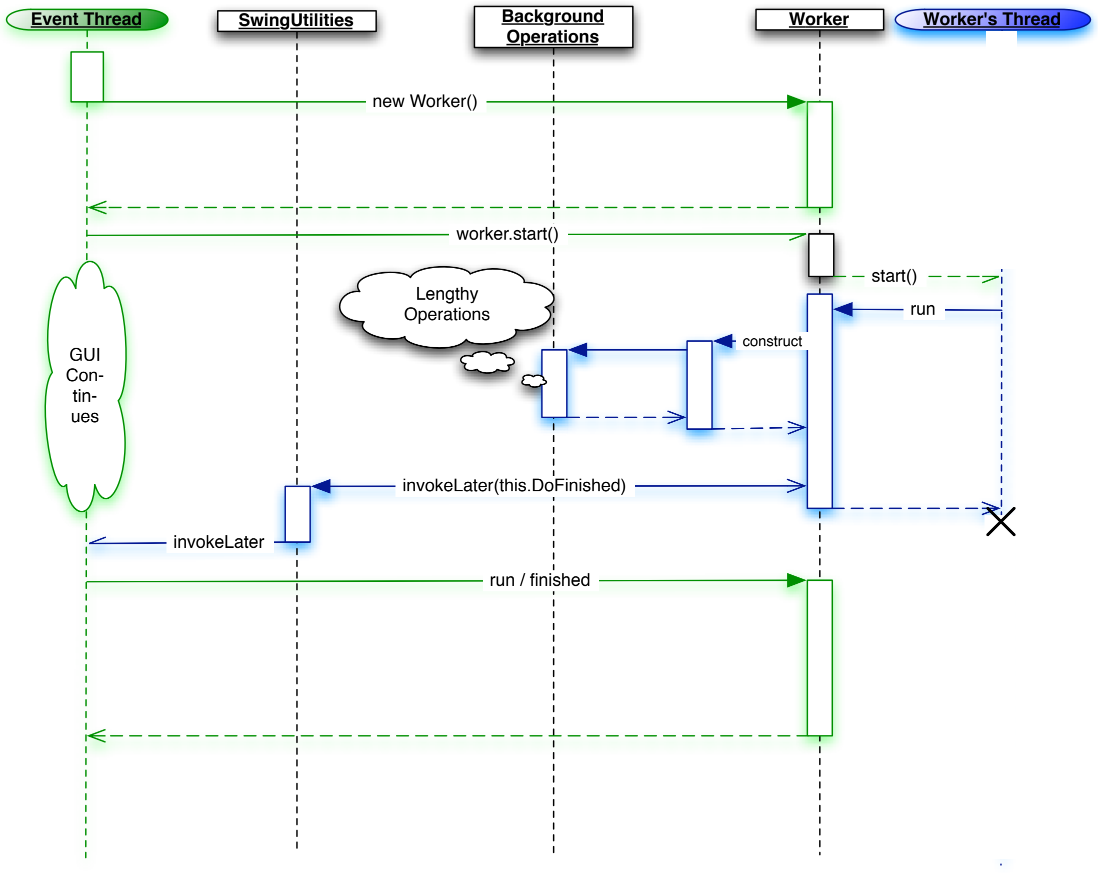


Figure 4.30: JHotDraw parallel `Worker` life-cycle

```
/**
 * @ColorImport java.awt.*
 * @ColorDeclare ProjDisabled, ProjWorker, FreeWorker
 * @IncompatibleColors ProjWorker, FreeWorker, AWT
 */
```

Figure 4.31: Thread coloring annotations for JHotDraw

```
/**
 * Executes the specified ProjectWorker on the worker thread of the project.
 * Execution is performed sequentially in the same sequence as the
 * ProjectWorkers have been passed to this method.
 *
 * @Color ProjDisabled
 */
public void execute(ProjectWorker worker) {
   if (executor == null) {
      executor = Executors.newSingleThreadExecutor();
   }
   executor.execute(worker);
}
```

Listing 4.3: JHotDraw ProjectWorker

a `ProjectWorker` thread when the `Project` is disabled. The color annotations on class `FreeWorker` similarly note that the `construct()` method will execute in a `FreeWorker` thread–that is, a thread that may be executing in parallel with other `Worker`s.

Now that we have created the new subclasses of `Worker` with their color annotations, we must change the **`new`** `Worker()` creations to use our new subclasses. Figure 4.33 on page 108 shows the new version of the typical sequential case. Immediately after the call to disable the `Project` on line 3 we have added a new lexical block, at which we grant the `ProjDisabled` color. Granting the color is appropriate, as we have just disabled the `Project`. Inside the block, we execute a **`new`** `ProjWorker()` (instead of the `Worker` in the old version). The `ProjWorker` inherits the color annotations discussed in the previous paragraph. The new lexical block ends immediately after the call to `project.execute`; the granted color expires along with the block. Once again, this is appropriate: the `Project` might still be disabled, but we do not know how long that will be true as the call to re-enable the `Project` will occur sometime after the `Worker` is finished. The key thing for this example is that we noted that the `Project` was disabled at the moment we invoked `project.execute`—which is exactly what we required in the in-line listing above. After modifying the other seventeen `Workers` in JHotDraw we can run the thread coloring analysis. The analysis's inference algorithm will propagate the `ProjDisabled`, `ProjWorker`, and `FreeWorker` colors around the call graph. We are now ready to actually *use* the colors to check something.

Using colors to check consistency in JHotDraw

At this point in the case study we can determine, for any given method, whether it is invoked from the `AWT` thread, a `Project`'s `Worker` thread, a parallel `Worker` thread, or none of the above. We have not, however, written more than a few color constraints on JHotDraw methods. Much of JHotDraw's code, unsurprisingly, is GUI code which inherits color constraints from the AWT and Swing frameworks.

The thread coloring analysis reports hundreds of errors on invocations of GUI-only methods from `ProjWorker` environments. What is happening here? My best guess is that the programmers expect that by disabling the `Project`, they have prevented the GUI from changing any data inside the `Project`. If this expectation is correct, they have temporarily made that data "immutable"—reading it would appear to be safe. There are, however, a few problems with this approach:

- Disabling the `Project` fails to disable all of the GUI parts inside the `Project`. Some other GUI frameworks recursively disable the contents of a container when the container itself is disabled. AWT and Swing do not do this. Did the programmers expect that disabling the `Project` would disable its contents as well? I can not tell from the available documentation and comments.
- JHotDraw depends on having each action that could ever change any portion of any `Project` remember to check whether that `Project` is disabled before making changes. If any action forgets to make the check, the assumption that disabling the `Project` makes its contents immutable (at least by the `AWT` thread) is broken.
- Even if the assumption that the `AWT` thread will not change the contents of disabled `Projects` is correct, the `Worker` threads should not modify `Project` contents either. But some of the `ProjWorker` threads are used to read in files and create images inside the `Project`. The Swing documentation is

```
public abstract class ProjectWorker extends Worker implements Runnable() {
 /** @Color (ProjDisabled & ProjWorker) */
 public abstract Object construct();
 ...
 /** @Color (ProjDisabled & ProjWorker) */
 public final void run() {
    final Runnable doFinished = new Runnable() {
       /** @Color AWT */
       public void run() {
          finished(getValue());
       }
    };
    try {
       setValue(construct());
    } catch (Throwable e) {
       e.printStackTrace();
    } finally {
       SwingUtilities.invokeLater(doFinished);
    }
 }
 ...
 public void start() {
    // Do nothing. Project workers should always run in the Project's executor thread.
    //new Thread(this).start();
 }
}

public abstract class FreeWorker extends Worker implements Runnable() {
 private Object value;
 /** @Color FreeWorker */
 public abstract Object construct();
 ...
 /** @Color FreeWorker */
 public final void run() {
    final Runnable doFinished = new Runnable() {
       /** @Color AWT */
       public void run() {
          finished(getValue());
       }
    };
    try {
       setValue(construct());
    } catch (Throwable e) {
       e.printStackTrace();
    } finally {
       SwingUtilities.invokeLater(doFinished);
    }
 }
 ...
 /** @Transparent */
 public void start() {
    new Thread(this).start();
 }
}
```

Figure 4.32: JHotDraw `Worker` class with color annotations

```
/** running in the AWT thread... */
 ...
 project.setEnabled(false);
 { //@Grant ProjDisabled
   // Open the file
     project.execute(new ProjWorker() {
        public Object construct() {
           try {
              project.read(file);
              return null;
           } catch (IOException e) {
              return e;
           }
        }
        public void finished(Object value) {
           ...
           setEnabled(true);
           ...
        }
     });
 }
```

Figure 4.33: JHotDraw typical sequential `Worker` usage with coloring

quite clear that the methods being invoked and fields being modified during this process should be touched only by the `AWT` thread. Even though the `Project` is disabled and JHotDraw's code will not modify the `Project` contents while the `Worker` is building those contents, the rest of Swing may still attempt to touch the newly created `Project` contents—perhaps even as the `Worker` is modifying those contents. I believe that this is clearly in error.

Next steps

Now that we have thread coloring annotations for the vast majority of the code in JHotDraw, what might we do next? Setting aside questions about AWT/Swing's thread usage policy and the `isEnabled()` pattern, we still have a variety of interesting options. We could:

- Figure out which actions are expected to run entirely in the `AWT` thread, and which are permitted to do background work. Armed with this information, we could enforce the policy using coloring.
- Identify data regions that should be touched only from the `AWT` thread, only from a `FreeWorker` or from a `ProjWorker` thread. Use data coloring to enforce the policies.
- Does JHotDraw contain methods that should be called only from one color environment or another? If we knew these thread usage policies, we could annotate the methods and let the thread coloring analysis enforce the policies.

Each of these options has the form "If we knew more about the designer's intentions we could enforce their intended policy." This shows us the limitations of a reverse-engineering case study performed without access to the original developers. Presumably, developers who are familiar with the project could express some of this missing policy information.

Coloring deficiencies

Our use of the `ProjectDisabled` color to restrict invocation of the `Project.Execute()` method and so prevent `Worker` thread actions when the `Project` is enabled seems unsatisfactory. We should not need to create subtypes of the `Worker` class in order to color the varieties of `Workers` correctly. The color constraints on the `run()` and `construct()` methods of those `Workers` should be fully checked against the color of the thread being created or used. A new feature—parameterized colors—may enable us to resolve these deficiencies. That said, we would also like to ensure that the `Project` whose `Worker` is being used is the same `Project` that was disabled. Our current prototype assurance tool does not attempt this check. Finally, our use of a thread color as a single global state flag appears rather like a poor man's version of Bierhoff and Aldrich's Typestates work [3], which may be a better fit for this aspect of our analysis of JHotDraw.

Other concurrency issues in JHotDraw

JHotDraw contains 10 synchronized keywords. Three of these are in a local implementation of a `LayoutManager` named `VerticalGridLayout`. Each of these synchronizes on the `lock "treeLock,"` which is an internal AWT/Swing lock. These appear to be reasonable. The other seven synchronized blocks are extremely questionable. In `AbstractCompositeFigure`, for example, methods `sendToFront` and `sendToBack` are synchronized, apparently to protect the list of child figures. If so, they fail completely as there are dozens of unprotected references to that list. The only possible sensible interpretation of synchronizing these two methods is as a policy lock to prevent attempting to execute both methods at once. All other interpretations involve possible state consistency errors.

The synchronization in JHotDraw is thoroughly suspect in my opinion. In every case that I have examined in detail, each datum that the synchronization might have protected also appears in non-synchronized contexts. I hypothesize that JHotDraw probably survives these errors because nearly all of the action takes place in the `AWT` event thread. As a result, race conditions will be infrequent because most of the time there is only one thread actually reading or writing the data. Background threads are invoked mainly for reading and writing files, and for printing. These are the places where race conditions would be most likely to appear.

4.7 JPL Chill Ground Data System case study

This case study shows that thread coloring SWT applications is as straightforward as thread coloring for AWT and Swing applications; specifically that most annotations for SWT GUI code are inherited from the libraries, just as with AWT and Swing. It then continues to address a particular concern of the Chill Ground Data System (Chill GDS) developers: they suspected that certain data regions are accessed only from the SWT event thread. If this single-thread access property were guaranteed, they wished to remove all synchronization for the data as part of a GUI performance enhancement. The study shows how we used data coloring to demonstrate that the particular data regions are in fact accessed only from the SWT event thread. The developers removed the needless synchronization and realized part of the performance improvement they sought.

My original goal for this in-field case study was to demonstrate that I can model and analyze the thread usage policy for SWT applications as well as AWT/Swing applications. In the event, this study also provided useful insight into Team and Developer ROI, the practicability of data coloring, our ability to produce results with minimal time and training of developers, and developer opinions about the thread coloring analysis, its expression, and its results.

4.7.1 The application

The Jet Propulsion Laboratory's "Chill GDS" ground data system ("Chill GDS" hereafter) serves two roles. In the first, it accepts incoming real-time data from spacecraft and rovers, and sends the data to one or more data stores (generally database applications) for later access. In its second role it acts as a GUI front-end through which users select the subset of available data they wish to access and route that data to their local data store for later scientific use. The users are generally science teams, some internal to NASA and others at universities and other collaborating organizations. Chill GDS is written using the SWT GUI framework. It has soft-real-time properties, in that it must sustain average data rates sufficient to serve the incoming data streams. The data streams are buffered before reaching the Chill GDS application, however, so Chill GDS need not achieve specific hard-realtime deadlines.

At the time of our case study, the Chill GDS developers were successfully meeting their performance goals when running headless. With the GUI turned on, however, they encountered substantial processing delays. Their goal for the case study was to remove unneeded synchronization in the hope of removing these delays. In the event, removal of unneeded synchronization provided a significant portion of the desired performance improvement. The remaining processing delays were not related to the issues discussed here, and are not discussed further.

```
1  /**
2   * @Promise "@ColorImport java.awt.*" for *
3   * @Promise "@ColorImport org.eclipse.swt.*" for *
4   * @ColorDeclare SWT
5   * @IncompatibleColors SWT, Compute, AWT
6   * @Promise "@Color SWT" for public **(**) &amp; !(asyncExec(*) | syncExec(*) | timerExec(*))
7   * @Promise "@Transparent" for public asyncExec(*) | syncExec(*) | timerExec(*)
8   */
9  package org.eclipse.swt;
```

Figure 4.34: Thread coloring annotations for SWT

4.7.2 Case study details

Coloring the SWT framework

To thread color a SWT application, we must first define the necessary thread colors, then place appropriate thread coloring annotations on the SWT framework code[11]. After that, we can begin the analysis of the client application. Thread coloring annotations were added to the framework and thread coloring analysis of those annotations with a toy example was performed at CMU well in advance of the case study in the field.

SWT follows the same general pattern of thread usage that we have seen in the AWT and Swing frameworks. There is at most one event thread at a time, along with zero or more other threads used for non-GUI work. The vast majority of SWT methods expect to be called only from the SWT event thread; a few are explicitly documented as allowing calls from any thread. The SWT documentation states clearly—but somewhat inconsistently—how the methods are distributed across these two categories. Specifically, the notation "`@throws SWTException with code ERROR_THREAD_INVALID_ACCESS`" appears in the JavaDoc for all non-widget SWT API methods that must run on the UI thread; similarly "Most methods on widgets must be called from the API thread" [21, pg 32]. Other documentation states that methods `asyncExec, syncExec` and `timerExec` are the only SWT methods that are permitted from non-UI threads. Examination of the source code shows that neither statement matches the as-written code. There are methods that may throw `SWTException` with code `ERROR_THREAD_INVALID_ACCESS` whose JavaDoc does not say so. There are methods whose documentation claims they should be called only from the UI thread, but whose implementation skips the threading check. That said, the second claim—that the three `...Exec` methods are the *only* SWT methods invokable from any thread—seems closest to the truth.

Figure 4.34 shows all of the color declarations required for annotation of the SWT framework.[12] Lines 2 and 3 are scoped promises that place a pair of `@ColorImport` annotations in every compilation unit in the SWT: one for the AWT color declarations, another for the SWT color declarations. On line 4 we declare the `SWT` thread color; it will be used for the SWT's UI thread. On line 5 we declare that the `SWT` color is distinct from the `AWT` and `Compute` colors; that is, no thread may be simultaneously associated with more than one of these three colors. On line 6 we write the scoped promise that applies the color constraint `SWT` to every public method in the SWT (with the exception of the three `...Exec` methods). Finally, line 7 states that the three `...Exec` methods are `@Transparent`; that is, they may be freely invoked from any thread.

Coloring JPL's Chill GDS

Now that we have annotated the SWT framework, we can begin on the Chill GDS application itself. The developers want to know whether they must synchronize access to a variety of data fields used in their GUI. Figure 4.35 on the facing page shows an example taken from class `EVRComposite`. The developers are *not* interested in analyzing thread usage of the non-GUI portions of their application. This allows us to use a thread coloring short-cut: we will work in terms of the `SWT` thread and "all other threads including unknown threads". That is, we do not apply coloring annotations to any non-GUI code. Having the tool report that it can not color something is sufficient to know that the something is not-GUI – more properly "not-GUI-*only*".

[11]Of course, we need not color the entire framework. The portions actually used by the application we care about will suffice.

[12]Lack of a module-scoped location for annotations actually requires us to repeat the "`@Color SWT`" scoped promise on line 6 in several additional packages. This is a temporary artifact of the current prototype assurance tool; it is not a fundamental requirement.

```
1  /** @Region protected evrComp
2   *  @ColorConstrainedRegions SWT for evrComp
3   */
4  protected class EVRComposite ... {
5   ...
6   /**
7    * @MapInto evrComp
8    * @Unique
9    * @Aggregate Instance into evrComp
10   */
11  protected Table evrTable;
12  ...
13  protected void createControls() {
14      ...
15      this.evrTable = new Table(...);
16      this.evrTable.setHeaderVisible(true);
17      ...
18      this.viewMenuItem.addSelectionListener(new SelectionListener() {
19         /** @-Color SWT */
20         //CMU: actually inherited from
21         // org.eclipse.swt.events.SelectionListener.widgetSelected()
22         // note that the '-' following the '@' turns the annotation off
23         public void widgetSelected(...) {
24            synchronized (evrTable) {
25               ...
26               ... = evrTable.getSelectionIndex();
27               ...
28               ... = evrTable.getItem(i);
29               ...
30            }
31         }
32      }
33      ...
34  } // end of createControls
35  ...
36 }
```

Figure 4.35: Selected portions of class `EVRComposite` from Chill GDS

Because most GUI methods inherit their color annotations from the GUI framework, we can move directly to coloring the data of interest to the developers.

For this case study we will answer the question: "Is the field `evrTable` from class `EVRComposite` accessed only from the `SWT` thread?" The Chill GDS development team sought answers to similar questions for many data fields in their GUI. In each case the analysis was essentially similar to the case we explore here. As usual with data coloring, we use Greenhouse-Boyland data regions to define the data of interest and Greenhouse-Boyland effects analysis to determine where that data is read or written.

We begin by declaring the data region we need. On line 1 of Figure 4.35 we declare a protected data region named `evrComp`. On line 2 we give a data coloring constraint: all access to region `evrComp` must happen from an `SWT` color environment; access from other color environments is in error. Next we must populate region `evrComp` with some data fields. The `@MapInto` annotation on line 7 notes that the region for field `evrTable` is part of region `evrComp`. The `@Unique` annotation on line 8 notes that there are no external references to the particular `Table` instance referred to by the `evrTable` field.[13] We need this annotation to ensure that we do not leak references to the `Table`. This property, when proven, permits much more precise effects analysis. The `@Aggregate` annotation on line 9 says that the instance data of the `Table` is also part of region `evrComp`, a property that depends on the uniqueness result from the previous annotation. All three of these annotations apply to the declaration of `evrTable` seen on line 11. These five annotations—four to set up the desired data region and one to establish the desired color constraint—are all we need to answer the developer's question. Figure 4.36 on the following page shows the tool's result for region `evrComp`: "All accesses to colorized region protected evrComp on EVRComposite are consistent with constraint SWT".

Let's look a little deeper. How did the tool know that this was the correct answer? First, we depend on the Greenhouse-Boyland effects analysis to find all references to the region of interest. In order to thread color this data, we compute an accessing color environment for the data region that is the union of the current color environment from each accessing program point. The accessing color environment represents all thread colors

[13]Greenhouse-Boyland uniqueness analysis can prove this property.

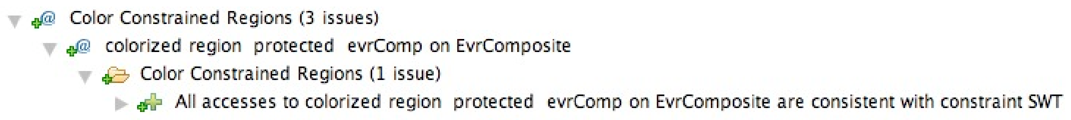

Color Constrained Regions (3 issues)
colorized region protected evrComp on EvrComposite
Color Constrained Regions (1 issue)
All accesses to colorized region protected evrComp on EvrComposite are consistent with constraint SWT

Figure 4.36: Data coloring result for region `evrComp`

that could ever touch the data region. We begin by considering the thread colors for the two methods shown in our example code fragment. Method `createControls()` is colored `SWT` by inference. That is, all calls to this method originate transitively from code that is known to be colored `SWT`. There are no `@Grant` or `@Revoke` annotations in `createControls()`, so the color environment throughout the method is simply `SWT`. Thus, the write to `evrTable` at line 15, the execution of the constructor at the same line, and the read from `evrTable` on line 16 all take place from the `SWT` thread.

The other references to `evrTable` seen in our code fragment (found on lines 24, 26 and 28) all take place within method `widgetSelected()`, which is known to be color `SWT` because it is an override of `org.eclipse.swt.events.SelectionListener.widgetSelected()`—a part of the SWT framework that is known to be `SWT`-only. In the absence of an explicit local annotation, the locally declared method inherits its parent's annotation. Once again, there are no `@Grant` or `@Revoke` annotations, so the entire body of the method takes place in an `SWT` color environment. References to `evrTable` found in other classes are handled similarly.

When we have analyzed all accesses to `evrTable`, we wind up with an accessing color environment of `SWT`. Since this environment satisfies the expressed constraint—$(\text{SWT} \Rightarrow \text{SWT}) = \mathit{true}$—our analysis concludes that the constraint on the data region is satisfied. The field `evrTable` is indeed accessed solely from the `SWT` thread. The Chill GDS development team can safely remove all synchronization for this field.

Our analysis of the data fields of interest to the developers showed that all of them were in fact accessed only from the `SWT` thread. Accordingly, they removed all synchronization on these fields and re-ran their test suite. All tests passed. This result is not surprising in hindsight. The source of the Chill GDS developer's suspicion that the fields of interest might be accessed only from the `SWT` thread was their discovery that the fields were not consistently locked. Some references were inside synchronized blocks as seen in Figure 4.35; others had no synchronization at all. The synchronization was thus ineffective: if it had not been redundant it would not have worked.

Developer reactions

The Chill GDS developers expressed satisfaction with and excitement about the results of the case study. They stated that they would have been afraid to remove their existing synchronization without the prototype assurance tool's analysis, even though they were fairly certain that the synchronization was broken. Having the analysis result showing that all access to the fields in question was confined to the `SWT` thread provided the necessary assurance to allow them to take this step. Some indicative quotes from the case study include:

"This would be too hard to sort out without a tool. I would never be certain I had analyzed all the cases."

"Annotation was easy once we sorted out the threading model."

"Can we keep the tool?"

"We should integrate this with our daily process."

"The tool's presentation of errors and results needs work. Each result makes sense, but it's too hard to find the ones I'm interested in among thousands of reports."

During informal discussions the developers indicated that they would like to use the analysis on a regular basis during their daily work. They did emphasize, however, that the tool's reporting was not well organized and needed work: "It's OK for research, but wouldn't fly if we'd paid money for it."

4.8 JPL 4Thread case study

In this case study I show the use of thread colors to track intended separation of the application into four important threads, plus a few additional minor threads. I then find by analysis that the developer misrepresented his thread usage model, and correct the model to reflect the reality of the as-written code. My original goal was to use thread coloring to verify the developer's understanding of which data are confined to individual threads and which are not. At the end of the case study the code was fully colored, and the developer and I were ready to begin using thread colors on the application's data. Unfortunately, we ran out of time before we were able to do any actual coloring of data.[14]

This case study demonstrates some anecdotal evidence on Team and Developer ROI: the developer invested an evening of his own time writing a complete set of color constraints for every method in his program. This recorded intent provided a significant advantage for the case study. We could not have colored the entire program in one afternoon without the developer's efforts.

4.8.1 The application

The "JPL 4Thread" application[15] is a "headless application"[16] that assists with action planning for spacecraft and rovers. Its input is a group of goals and some subset of the actions required to accomplish each individual goal. The 4Thread application is divided into four important threads: the `NetworkProposer`, the `XGoalChecker`, the `NetworkExecutor`, and the `NotificationQueue`. There are a number of additional threads that are not considered important. The intended operation of the application is that the `NetworkProposer` creates networks containing multiple goals and actions, and proposes specific possible choices of network. The `XGoalChecker` checks something about these networks, and selects the 'best choice' by some criteria not known to me.[17] The `NetworkExecutor` performs a symbolic execution of the network, placing events (also called *commands*) into the `NotificationQueue`. The contents of the `NotificationQueue` are later used to operate the spacecraft or rover.

There is one `NetworkProposer` thread, one `XGoalChecker` thread, one `NetworkExecutor` thread, one `NotificationQueue` thread, and one everything-else thread. These threads are carefully separated, partly to gain performance by having multiple threads working at once, and partly to enhance separation of concerns and to reduce coupling. The developer indicates that future versions of this program may have more than one `NetworkProposer` thread and more than one `NetworkExecutor` thread.

4.8.2 Case Study Details

The developer's original description of the application's thread usage policy had a strict partition of code such that each of the four main threads had several classes that expect to be called only from that thread. We therefore begin by defining colors for each of the threads and granting those colors at the `run()` method for each thread. Figure 4.37 on the next page shows the annotations used. On line 2 we see the declarations of the thread colors we need for our initial threading model. The first four are the important threads. The `Tester` thread is used only for unit testing; the `Initializer` thread runs only at start up. On line 4 we declare that no thread may simultaneously be associated with more than one of these colors. If this application had a GUI, we would add the `AWT` color to the list of IncompatibleColors; this would indicate that the GUI event thread is not one of these threads either. On lines 6 through 9 we declare that there may be at most one thread of each important color at any given time.[18] The thread color declarations on line 10 are not part of the initial model.

Now that we have declared the necessary thread colors, we must grant them at suitable program points. Figure 4.38 on the following page shows selected portions of the code for the `XGoalChecker` thread. At line 5 we see the creation of the `Thread` that will be the `XGoalChecker` thread. We insert the color annotation on line 6 to indicate that the `run()` method will execute only on the `XGoalChecker` thread. The declarations, creation, and coloring of the other four major threads are essentially similar.

The developer of the 4Thread application was so excited about the idea of thread coloring that he wrote color constraints for all of his methods the night before the case study. Figure 4.40 on page 115 shows extracts

[14]This case study was performed during one afternoon of a three-day visit to the Jet Propulsion Laboratory.

[15]Not its real name, at the developer's request.

[16]That is, it does not have a user interface.

[17]Fortunately, I don't need to know the criteria for the purposes of the case study.

[18]Remember, the `@MaxColorCount` annotation is purely declarative; it is not checked for correctness.

```
/**
 * @ColorDeclare NetworkProposer, NetworkExecutor, XGoalChecker, NotificationQueue,
 *   Tester, Initializer,
 * @IncompatibleColors NetworkProposer, NetworkExecutor, Tester,
 *   Initializer, XGoalChecker, NotificationQueue
 * @MaxColorCount NetworkProposer 1
 * @MaxColorCount NetworkExecutor 1
 * @MaxColorCount XGoalChecker 1
 * @MaxColorCount NotificationQueue 1
 * @ColorDeclare NetworkPred, GoalCheckPred, NQPred ...
 * ...
 */ package ...;
```

Figure 4.37: Thread coloring annotations for JPL 4Thread application

```
public class XGoalChecker implements ... {
 ...
 private XGoalChecker() {
    ...
    xGoalCheckerThread = new Thread( new Runnable() {
       /** @Color XGoalChecker */
       public void run() {
          ...
          while ( true ) {
             ...
             XGoal xg = ...;
             ...
             assert xg.isExecuting();
             ...
          }
       }
       ...
    });
 }
 ...
}
```

Figure 4.38: Body of `XGoalChecker` from JPL 4Thread application

```
public abstract class GoalNetworkElement ... {
 ...
 /** @Transparent */
 protected boolean isExecuting() {
    return ... Network.getExecutingNetwork() ... ;
 }
}
```

Figure 4.39: Example predicate method from JPL 4Thread application

```
 1 public class Network extends ... {
 2   ...
 3   /** @Color (NetworkProposer | Initializer | Tester) */
 4   public static Network initialize( ... ) ... { ... }
 5
 6   /** @Color Initializer */
 7   public static void waitForReadyToExecute() { ... }
 8
 9   /** @Color NetworkProposer */
10   public static Architecture getArchitecture() { ... }
11   ...
12   /** @Color (NetworkProposer | NetworkExecutor) */
13   /*package*/static Network getExecutingNetwork() { ... }
14   ...
15   /** @Color NetworkProposer */
16   public void promote( RtEpoch now, Duration planningOverhead ) throws MDSException { ... }
17
18   /** @Color NetworkProposer */
19   /*package*/synchronized void delete() { ... }
20
21   /** @Color (NetworkProposer | NetworkExecutor) */
22   public boolean isExecuting() { ... }
23
24   /** @Color NetworkProposer */
25   public boolean isReadyToExecute() { ... }
26   ...
27 }
```

Figure 4.40: `Network` class from JPL 4Thread with color constraints written by the original developer

from one of his annotated classes. Note in particular that each method is annotated to allow invocation only by the expected threads. These annotations embody the developer's understanding of his thread usage model.

The clean distinction between the four major threads turned out to be illusory. In fact, the code contains a group of basic predicate functions such as `GoalNetworkElement.isExecuting()` from line 4 of Figure 4.39 on the facing page. These predicates are intended for use from any thread. Many of the predicate methods are implemented using methods that "should be called only by the `NetworkProposer` thread" for example. Indeed, the implementation of `GoalNetworkElement.isExecuting()` includes at line 5 an invocation of `Network.getExecutingNetwork()` (line 13 of Figure 4.40). But that latter method is annotated as being callable only from the `NetworkProposer` or the `NetworkExecutor` threads (line 12). Our initial attempt at thread coloring analysis reported dozens of errors of this kind. Clearly the developer's thread usage model does not match the as-written code in this respect.

Analysis

It is interesting to note that the developer of the JPL 4Thread application is one of the most sophisticated developers the Fluid group has encountered during the course of our case studies. He has participated in every diagnostic visit we have made to JPL. We have analyzed the locking and concurrency behavior of four different programs he has written, all without ever finding a threading-related bug. This is exceedingly unusual in our experience. Because of the developer's demonstrated sophistication we were quite surprised to find that his stated thread usage model failed to reflect his as-written code. One possible lesson to draw from this experience is that even top-notch developers forget the details of programs they have written.

Fixing the color model

On investigation, the developer indicated that the "real truth" is that the predicates have been carefully written to use combinations of calls and arguments that are "O.K. from any thread" and so should be permitted to invoke the "thread-private" methods—even though the predicates may be executing in a thread that is not the "home thread" of the thread-private methods. In the case of `Network.getExecutingNetwork ()` on line 13, his concern is that other threads should not be permitted to grab hold of a reference to a `Network`, hold on to that reference, and potentially peek and poke at its fields in an uncontrolled fashion. Predicate functions like `GoalNetworkElement.isExecuting()`, however, are notionally provided by the `Network` implementation for the use of those other classes, and should be permitted to make internal use of methods such as `Network.getExecutingNetwork()`. The predicates have been "carefully coded so as not to leak information and so as to perform only safe operations." This thread usage pattern is pervasively present throughout the 4Thread application.

We now embark on expressing this more complicated model. Our goal is to preserve the declaration that the predicate functions are `@Transparent`, that is that they may be freely invoked from any thread. Meanwhile, we desire to maintain the restriction that arbitrary threads may not invoke `Network` methods directly (for example), unless they do so by way of a predicate function.

Our approach is to create new thread colors for each thread's exported predicate functions. These annotations can be seen at line 10 of Figure 4.37 on page 114. Each predicate function continues to be declared as `@Transparent` (see for example line 3 of Figure 4.39 on page 114), so any thread may invoke it. Meanwhile, inside the body of the predicate functions we grant the appropriate new predicate color. In the case of `GoalNetworkElement.isExecuting()` this would require the addition of the annotation `@Grant NetworkPred` at the end of line 4.[19] To complete the model, we must change the coloring annotations of all the methods used to implement these predicates—directly or transitively—to permit invocation from color environments that include the new `...Pred` colors. For `Network.getExecutingNetwork()` this would mean changing its color constraint from that shown on line 12 to */∗∗ @Color (NetworkProposer | NetworkExecutor | NetworkPred); ∗/* instead. For methods that are not user annotated, the tool will infer the correct annotation. Methods that are user annotated will require changes like that for `Network.getExecutingNetwork()`. The tool will indicate these methods by reporting coloring errors for them until their annotations are corrected. Once the new model is fully in place we then have a clear distinction between methods that

- Are truly called only within "their own thread"
- Might also be called from another thread via a predicate function
- May be called from some other combination of threads.

The new model also enables data coloring to distinguish data that are truly thread-local from data that are potentially shared due to being transitively accessed from predicate functions as well as from thread-private methods.

Analysis

Thread coloring the data was the developer's initial goal for the case study. Due to the need to re-do his initial thread usage model, we ran out of time and did not get that far. On the other hand, we did succeed in:

- Demonstrating that his initial thread usage model was incorrect,
- Developing a corrected thread usage model for the application, and
- Verifying that all methods executed only in the expected threads according to the corrected model.

Furthermore, another half-day of work would likely have been sufficient to provide some of the data coloring results the developer was looking for.

4.9 Conclusion

Both the thread coloring analysis and its prototype implementation have been quite successful. The case studies presented in this chapter collectively address the practicability issues I set out to investigate. Firstly, the analysis and its implementation do indeed produce results that address the mechanical properties of thread usage. Secondly, the case studies show that thread coloring for code is a relatively straight-forward and manageable process even when, as for all of the case studies presented here, one is working with other people's code. Further, I have demonstrated successful results using data coloring (see the JPL Chill GDS case study, Section 4.7 for details). In that in-the-field case study, the developers were primarily interested in determining whether certain data regions of interest were confined to the SWT event thread. Once the tool

[19]It may seem odd that we permit granting a color in the context of a `@Transparent` annotation. Such `@Grant` annotations are permitted if and only if the granted color has no color conflicts—a condition which ensures that the `@Grant` can never introduce a coloring problem regardless of the color environment from which the `@Transparent` method was invoked.

showed this through analysis, the developers removed all synchronization from the data regions of interest, achieved much of the performance increase they sought, *and* passed their regression tests. They commented that without the tool's analysis they would not have been willing to remove synchronization, for fear of introducing a race condition or other threading-related bug. Data coloring was also the original goal of the JPL 4Thread case study (see Section 4.8 on page 112); with only four hours available to work in we ran out of time before we were able to get that far.

The thread coloring language and its supporting annotations have been sufficiently expressive to model every thread usage policy encountered to date, with one exception—the current implementation does not fully support `Runnable`s and other methods for which substitutability of thread color constraints is an issue. The work-around for this issue is no worse than a cast. See Section 5.3.1 on page 130 for further discussion of the issue. Non-substitutable methods aside, I have been able to model thread usage policies and color other people's code with no major problems. I did, however, encounter two problems that were not directly related to the coloring process.

Firstly, in some case studies, such as the JHotDraw case study (see Section 4.6 on page 100), the process of thread coloring was handicapped by a lack of information about the intended thread usage policy. This does not indicate a deficiency of the thread coloring analysis. Rather, it demonstrates a limitation of reverse engineering design intent from unannotated code. The reverse engineering process may fail. Even when successful, reverse engineering requires substantial time and energy.

Secondly, thread coloring of an application requires the existence of annotated thread usage models for the libraries and frameworks used by that application. The annotations that express these models must be written by some developer. Because most libraries and frameworks are used by many client applications, and because the annotations for those libraries and frameworks need be written only once, we do not count the cost of annotating the libraries as part of the cost of annotating client applications. We hope that library and framework authors will see thread coloring annotations as a way to provide added value to their clients.

Three of the case studies shed light on issues of scale. The Electric case study (see Section 4.5 on page 90), at roughly 140KSLOC, provided an excellent test of the prototype assurance tool's ability to handle medium-sized programs. The low numbers of annotations required for complete thread coloring of Electric's code (6.3 annotations per KSLOC) suggest that the effort required to color programs of this size should be manageable. The even lower numbers we believe to be achievable—0.25 per KSLOC—suggest that the analysis can scale to much larger applications with reasonable end-user effort. The Sky View Café case study (see Section 4.4 on page 84), at roughly 25KSLOC, was originally selected to serve as the test of scale in program size. Instead, it served to stress scale in terms of numbers of thread roles and complexity of thread usage policy. The JPL 4Thread case study also involved a complex thread usage model. Thread coloring and its prototype implementation met each of these scaling challenges successfully.

Case studies in the field provided the opportunity to gather anecdotal information on Team and Developer ROI, as well as on achieving results in limited time and with minimal training. Every development team the Fluid group has worked with wanted to continue using the prototype assurance tool as part of their regular development process. Most of the individual developers stated that they would eagerly do so as well. These opinions suggest that we may have succeeded in addressing Team and Developer ROI. The limited time available during in-the-field case studies provided useful information about training needs, about developer engagement with the tool, and about developer comprehension of the annotations used for model expression. Sophisticated and experienced developers, such as the author of the JPL 4Thread application, were able to write their own thread coloring annotations after about fifteen minutes' training. Less sophisticated developers did not write their own annotations, but were easily able to describe verbally "what the annotation should say" using the terminology of thread coloring. Every developer, including the creator of the thread coloring analysis, struggled with finding the analysis results of interest in the prototype tool's voluminous output. This area remains a fertile ground for improvement.

Chapter 5

Formal model

5.1 Introduction

This chapter provides a rigorous definition of the static semantics of thread coloring. This rigorous definition serves several purposes. Developing the formal model makes precise the basis of thread coloring and provides a foundation for the development of the key analysis algorithms. The analysis tool development was informed by this model definition.

The model is also an initial step towards a proof of soundness. That proof requires this structural model to be augmented with a dynamic semantics. The proof and the dynamic semantics are deferred to future work, though we do sketch the approach in the final section of this chapter.

This chapter begins by defining the language "Featherweight Java with Colors" (FJC), based on Featherweight Java by Igarashi *et al.* [31] (often referred to as FJ). It continues with a description of a somewhat expanded model of the thread coloring language, followed by a discussion of the differences between FJC and the full Java language with the full thread coloring language.

5.2 Featherweight Colors

We define a language "Featherweight Java with Colors" (referred to as FJC), based on Featherweight Java (FJ) by Igarashi *et al.* FJC encompasses the same limited subset of the full Java language as does FJ. The reduction in supported Java features necessitates some limitations on the thread coloring language. FJC expresses: assignment of roles to threads, dynamic changes of assigned roles, and constraints both on multiplicity of threads and on simultaneity of roles. Roles are expressed using their names as uninterpreted tags, which we call colors. The thread coloring additions to FJ enable us to model thread usage policies and to reason about the consistency of that model with FJ code.

Our model consists of two parts:

- Color bindings, which associate a set of color names with each thread
- Color constraints, that restrict which threads may execute a segment of code or access a region of state.

A color constraint is denoted as a Boolean expression over color names. It expresses the powerset of the acceptable color bindings that may be associated with a particular thread that executes a segment of code or accesses a region of state. A consistently colored program is one in which no color constraint will ever be violated on any execution of the program.

5.2.1 Abstract syntax

Our syntax for Featherweight Java with Colors (FJC) is the same as that for FJ, with three exceptions. The major difference is our addition of color constraints and `Grant` and `Revoke` annotations (along with the syntax for the Boolean expressions used in the constraints, of course). We support constraints for fields, methods, and constructors. The idea behind these constraints is to restrict access to the fields or invocation of the methods or constructors to only those portions of a program whose color environment (see Section 5.2.3 below) satisfies the constraint.

One minor difference appears in our treatment of sequences. The length of a sequence $\overline{\mathrm{x}}$ is written $|\overline{\mathrm{x}}|$ in FJC, to more closely follow standard notation. As in FJ, we write $\overline{\mathrm{f}}$ as shorthand for $\mathrm{f}_1, \ldots, \mathrm{f}_n$ where $\mathrm{n} \geq 0$ (and similarly for the other metavariables). Unlike FJ, we abbreviate sequences of pairs and triples by writing "$\overline{\mathrm{C}\,\mathrm{f}}$" as shorthand for "$\langle \mathrm{C}_1\ \mathrm{f}_1 \rangle, \langle \mathrm{C}_2\ \mathrm{f}_2 \rangle, \ldots, \langle \mathrm{C}_n\ \mathrm{f}_n \rangle$" and "$\overline{\mathrm{C}\,\mathrm{f}\,\mathrm{c}}$" as shorthand for "$\langle \mathrm{C}_1\ \mathrm{f}_1\ \mathrm{c}_1 \rangle, \langle \mathrm{C}_2\ \mathrm{f}_2\ \mathrm{c}_2 \rangle, \ldots, \langle \mathrm{C}_n\ \mathrm{f}_n\ \mathrm{c}_n \rangle$".

The final difference between FJ and FJC appears in our definition of constructors. To increase clarity, we write $\mathrm{C}\left(\overline{\mathrm{C}\,\mathrm{g}},\ \overline{\mathrm{C}\,\mathrm{f}}\right)$ to distinguish the arguments consumed by the given constructor from those passed on to `super()`. We denote the arguments consumed by the given constructor as $\overline{\mathrm{C}\,\mathrm{f}}$; the arguments passed to `super()` are denoted as $\overline{\mathrm{C}\,\mathrm{g}}$.

We use metavariables in the style of FJ[1].

Table 5.1: Metavariable Usage

Metavariables	Range
A, B, C, D, E	Class names
f, g, h	field names
m	method names
x,y	parameter names
d, e	expressions
CL	class declarations
K	constructor declarations
M	method declarations
•	Empty sequence, empty expression
c, cc,ccc	color constraints (added for FJC)
r	color revokes (added for FJC)
a	color grants (added for FJC)
cn, cname	color names (added for FJC)
cExp	Boolean expression over color names (added for FJC)

As in FJ, we denote concatenation of sequences using a comma.

The syntax for FJC is:

$$\begin{array}{llll}
(\text{new for FJC}) & \mathrm{a} & ::= & \texttt{Grant}\ \overline{\mathit{cname}} \\
 & \mathrm{r} & ::= & \texttt{Revoke}\ \overline{\mathit{cname}} \\
 & \mathrm{c} & ::= & \texttt{Color}\ \mathrm{cExp} \\
(\text{modified for FJC}) & \mathrm{CL} & ::= & \texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{C}\ \left\{\overline{\mathrm{C}\,\mathrm{f}\,\mathrm{c}};\ \mathrm{K}\,\overline{\mathrm{M}}\right\} \\
 & \mathrm{K} & ::= & \mathrm{C}\left(\overline{\mathrm{C}\,\mathrm{g}},\ \overline{\mathrm{C}\,\mathrm{f}}\right)\ \mathrm{c}\ \left\{\bar{\mathrm{r}};\ \bar{\mathrm{a}};\ \texttt{super}\,(\bar{\mathrm{g}})\ ;\ \texttt{this}.\bar{\mathrm{f}} = \bar{\mathrm{f}}\,;\right\} \\
 & \mathrm{M} & ::= & \mathrm{C}\ \mathrm{c}\ \mathrm{m}\left(\overline{\mathrm{C}\,\mathrm{x}}\right)\{\bar{\mathrm{r}}\,;\bar{\mathrm{a}}\,;\ \texttt{return}\,\mathrm{e}\,;\} \\
(\text{unchanged from FJ}) & \mathrm{e} & ::= & \mathrm{x} \\
 & & | & \mathrm{e.f} \\
 & & | & \mathrm{e.m}\,(\bar{\mathrm{e}}) \\
 & & | & \texttt{new}\ \mathrm{C}\,(\bar{\mathrm{e}}) \\
 & & | & (\mathrm{C})\,\mathrm{e}
\end{array}$$

where $\overline{\mathit{cname}}$ is a sequence of color names, and *cExp* is a Boolean expression over color names (syntax below). By convention the keywords `Grant`, `Revoke`, and `Color` may be omitted if their following sequence or expression is empty (*i.e.*, there is no need to write Grant •).

For convenience, and without loss of generality, all color expressions—including color environments (see below)—are expressed in disjunctive normal form (DNF). The syntax below for Boolean expressions in DNF was added for FJC. The construct $\{\mathrm{term}, \wedge\}^+$ denotes "one or more instances of a term separated by $\wedge$ operators."

[1]We follow the notation from FJ rather loosely, as we have chosen to make quite a few clarifying changes.

$$
\begin{array}{lll}
\text{cExp} & ::= & cn \\
 & | & \text{conjClause} \\
 & | & \left(\{\text{conjClause},\vee\}^{+}\right) \\
\text{conjClause} & ::= & \text{term} \\
 & | & \left(\{\text{term},\wedge\}^{+}\right) \\
\text{term} & ::= & cn \\
 & | & \neg cn
\end{array}
$$

When operators other than $\wedge, \vee,$ and $\neg$ appear in a Boolean expression, we mean that the operator—$\exists$, for example—is applied to the expression, and the resulting Boolean expression is then simplified to a logically equivalent formula that is in DNF.

FJC supports a quite restricted subset of Java. As with the underlying FJ language, we omit concurrency, inner classes, generics, reflection, assignment, interfaces, overloading, messages to `super`, `null` pointers, base types, abstract method declarations, shadowing of fields, access control, and exceptions. Features that are part of both FJ and FJC include mutually recursive class definitions, object creation, field access, method invocation, method override, method recursion through `this`, sub typing and casting. The only form of assignment supported is the initialization of fields in object constructors. This simplified Java is small enough to model easily, while providing sufficient richness to allow a clear explanation of thread coloring.

It may seem surprising that we analyze threading properties of code using a model that supports neither concurrency nor threads. Fortunately, our thread coloring analysis can avoid considering threads directly. All color constraints are expressed in terms of a restriction on the color bindings *of the current thread*. The color environment represents the possible color bindings of all threads that may execute at the program point where the color constraint is being checked. Thus, the analysis can treat threads as just another type, whose methods and fields are handled exactly in the usual fashion.

An FJC program is a collection of class definitions together with an expression to be evaluated and an initial color environment. Additionally, each program has associated a set of color names Col and a set of class names Cls, which are the universe of color and class names used in the program. Class names also serve as type names because FJC—like the underlying FJ—has no interfaces. We abuse notation by writing $\mathrm{m} \in \mathrm{Cls}$ to denote "some method in the program," and likewise for class names, class declarations, constructors, and the like (using appropriate meta-variables).

Example

We use the code in Figure 5.1 as a running example. It is the example from Igarashi *et. al.*, [31], extended by adding color constraints to several of its methods and fields.

5.2.2 Computation semantics and expression typing

FJC changes neither the computation semantics nor the expression typing of Featherweight Java. Instead, we add a new judgment about whether the color environment satisfies the expressed color constraints, a procedure described below. It is not necessary to examine details of semantics and typing judgments for purposes of this exposition, which is intended to be self-contained. We do, however, assume that all FJC programs are correctly typed according to the semantics of FJ.

We refer to several FJ constructs in support of our judgments about color environments. A type environment Γ, called an "environment" in FJ, is a finite mapping from variables to types, written $\overline{\mathrm{x}} : \overline{\mathrm{C}}$. The typing judgment for expressions in FJ, and thus FJC, has the form $\Gamma \vdash \mathrm{e} \in \mathrm{C}$, read "in the type environment Γ, expression e has type C."

5.2.3 Color environment

A color is an uninterpreted tag that represents a particular role (or purpose) for a thread. A thread can have zero or more colors at any given program point or data reference. The binding between a thread and its

```
class A extends Object {
  A() color c4 {revoke c4, c3; super(); }
}

class B extends Object {
  B() { super(); }
}

class Pair extends Object {
  Object fst color c3;
  Object snd;
  Pair(Object fst, Object snd) color c3 {
    grant c1; super(); this.fst=fst; this.snd=snd;
  }
  Pair setfst(Object newfst) color c3 {
    return new Pair(newfst, this.snd);
  }
}
```

Figure 5.1: Running example program

colors—called a *color binding*—is a set of colors. Colors that are members of the set are associated with the thread, all other colors are not. Color bindings are not fixed; they can change during the execution of the thread. Color bindings could be represented by a bit-vector with one bit for each color in the color universe. In such a vector, the bit for a color is set when the thread and the color are associated, and clear when they are not.

During a given execution at a particular program point, a given thread has some particular binding. At another time during the same execution, the same thread may have a different binding at the same program point. Likewise, some other thread may arrive at that program point with either the same or a different binding. A *color environment* χ (pronounced *Ki*, taken from the first letter of *chroma*, the Greek word for *color*) is the set of all permissible color bindings at a particular program point, for all possible executions of the program. It is thus a set of sets of colors; the color binding of any thread that reaches that program point must be an element of the color environment. We can represent a color environment as a Boolean expression with color names as variables, whose extension is the desired set of color bindings. Thus, the single expression $(\mathrm{c4} \vee \neg\mathrm{c2})$ represents a single color environment whose bindings include $\{\mathrm{c4}\}$, $\{\}$, $\{\mathrm{c4}, \mathrm{green}\}$ and all other color bindings that include $\mathrm{c4}$ or that do not include $\mathrm{c2}$. For example, if the color universe is $\{\mathrm{c4}, \mathrm{c2}, \mathrm{blue}, \mathrm{green}\}$, this expression represents twelve of the sixteen possible color bindings.

A coloring judgment for an expression, written as

$$\chi \vdash \mathrm{e}$$

is read "the color environment χ satisfies the constraints on the expression e." For example, given the above definitions the expression

$\mathrm{c4} \wedge \mathrm{c3} \vdash$ `new Pair(new A(), new B()).setfst(new B())`

is consistently colored[2]. If we had omitted the color environment, we would have been unable to judge the color consistency of the expression.

Auxiliary definitions

In order to define color consistency, we need a few auxiliary definitions. In these equations we follow the conventional notation in which premises are seen above the bar and conclusions are seen below the bar.

Field lookup: The fields of a class C, written *fields* (C) are a sequence of triples $\langle \mathit{type}, \mathit{name}, \mathit{constraint} \rangle$ for all the fields declared in class C and in any of its super-classes. Thus the helper function *fields* has the classes as its domain and produces results whose range is triples as defined above.

[2]This example is worked out in full below.

$$\frac{}{\mathit{fields}\,(\texttt{Object}) = \bullet} \tag{5.1}$$

$$\frac{\mathit{CL}(\mathrm{C}) = \texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{D}\ \left\{\overline{\mathrm{E\ f\ c}};\ \mathrm{K}\ \overline{\mathrm{M}}\right\} \quad \mathit{fields}\,(\mathrm{D}) = \overline{\mathrm{A\ g\ cc}}}{\mathit{fields}\,(\mathrm{C}) = \overline{\mathrm{A\ g\ cc}},\ \overline{\mathrm{E\ f\ c}}} \tag{5.2}$$

This is a straightforward extension of the field look up function from Featherweight Java. All we have done is to add the color constraints for fields to the system. For example, using the example program in Figure 5.1,

$$\mathit{fields}\,(\mathrm{Pair}) = \langle \mathrm{Object,\ fst,\ c3} \rangle\,, \langle \mathrm{Object,\ snd,}\ \bullet \rangle$$

This result mirrors that from [31], except that the FJC field lookup function produces a sequence of triples rather than FJ's sequence of pairs. The type and name members of each triple are the same as those for the pairs in FJ; we have added the color constraint for FJC. Because Object has no fields, it cannot contribute to the list.

Color constraint expressions: The expression of a color clause c, written $\mathit{expr}\,(\mathrm{c})$, is the Boolean expression, if any, from a possibly absent color clause.

$$\frac{\mathrm{c} = \mathrm{color}\ \mathit{cExpr}}{\mathit{expr}\,(\mathrm{c}) = \mathit{cExpr}} \tag{5.3}$$

$$\frac{\mathrm{c} = \bullet}{\mathit{expr}\,(\mathrm{c}) = \bullet} \tag{5.4}$$

Thus, *expr* is a function whose domain is color clauses, and whose range is Boolean expressions.

Constraint look up: The helper function *cnstr* has a domain of field references, method invocations, and constructor invocations; its range is Boolean expressions. This function, written $\mathit{cnstr}\,(\mathrm{e.f})$ (for example), yields a Boolean expression over color names that must be satisfied by the color environment in which it appears. The color constraint on a Java entity—whether method, field, or constructor—defines the maximum environment that is acceptable for using that entity. That is, the color environment from which the entity is invoked or accessed must be a subset of the environment defined by the constraint expression.

For fields:

$$\frac{\Gamma\vdash \mathrm{e}_0 \in \mathrm{C}_0 \quad \mathit{fields}(\mathrm{C}_0) = \overline{\mathrm{C\ f\ c}}}{\mathit{cnstr}\,(\mathrm{e}_0.\mathrm{f}_i) = \mathit{expr}\,(\mathrm{c}_i)} \tag{5.5}$$

The premise $\Gamma\vdash \mathrm{e}_0 \in \mathrm{C}_0$ should be read as "the type of expression e_0 is C_0," so (5.5) says that the constraint of a field selection expression is the constraint of the field defined in the class found from the type of the expression—exactly as one would expect.

For example, in the context of our example program in Figure 5.1,

$$\mathit{cnstr}\,(\mathrm{Pair.fst}) = \mathrm{c3}$$

and

$$\mathit{cnstr}\,(\mathrm{Pair.snd}) = \bullet$$

In other words, the color constraint of the fst field from class Pair is c3. Likewise, Pair.snd has no color constraint.

For method invocation:

$$\frac{\mathit{CL}(\mathrm{C}) = \texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{D}\ \left\{\overline{\mathrm{E\ f\ cc}};\ \mathrm{K}\ \overline{\mathrm{M}}\right\} \quad \mathrm{B\ m\ c}\ \left(\overline{\mathrm{A\ x}}\right)\ \{\bar{\mathrm{r}};\ \bar{\mathrm{a}};\ \texttt{return}\ \mathrm{d};\} \in \overline{\mathrm{M}} \quad \Gamma \vdash e_0 \in \mathrm{C}}{\mathit{cnstr}\,(\mathrm{e}_0.\mathrm{m}\,(\bar{\mathrm{e}})) = \mathit{expr}\,(\mathrm{c})} \tag{5.6}$$

so

$$\mathit{cnstr}\,(\mathrm{Pair.setfst}\,(\mathrm{new\ B}\,())) = c3$$

Equation

For `new` expressions (a.k.a. constructor invocations):

$$\frac{\begin{array}{c} \mathit{CL}(\mathrm{C}) = \texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{D}\ \left\{\overline{\mathrm{B\ f\ cc}}\,;\ \mathrm{K}\ \overline{\mathrm{M}}\right\} \quad \mathit{fields}\,(\mathrm{D}) = \overline{\mathrm{A\ g\ ccc}} \\ K\ =\ C\left(\overline{\mathrm{A\ g}},\ \overline{\mathrm{B\ f}}\right)\ \mathrm{c}\ \left\{\overline{\mathrm{r}}\,;\ \overline{\mathrm{a}}\,;\ \texttt{super}\,(\overline{\mathrm{g}})\ ;\ \texttt{this}.\overline{\mathrm{f}} = \overline{\mathrm{f}}\,;\right\} \end{array}}{\mathit{cnstr}\,(\texttt{new}\ \mathrm{C}\,(\overline{\mathrm{e}})) = \mathit{expr}\,(\mathrm{c})} \tag{5.7}$$

so in the context of our example program,

$$\begin{array}{c} \mathit{cnstr}\,(\texttt{new}\ \mathrm{A}\,()) = \mathrm{c1} \\ \mathit{cnstr}\,(\texttt{new}\ \mathrm{B}\,()) = \bullet \end{array}$$

Color consistency: In order to determine whether an expression (or method, or constructor) is consistently colored, we must determine whether the color environment satisfies all relevant constraints on the expression. The rules that follow define color satisfaction for each of the syntactic expression kinds.[3]

We now introduce some additional notation. The color environment χ is a set of sets of colors, which we can represent using a Boolean expression. The notation $\chi \Rightarrow \mathit{SomeBooleanExpression}$ can be read either as "the Boolean expression that represents χ implies SomeBooleanExpression" or as "χ is contained in the set of sets reified by SomeBooleanExpression;" either definition conveys the necessary property. The color consistency judgment for expressions has the form $\chi \vdash \mathrm{e}$, read "in the color environment χ, expression e is consistently colored. The color consistency rules for expressions are syntax directed, with one rule for each form of expression.

$$\frac{}{\chi \vdash x} \tag{5.8}$$

Simple variables do not have color constraints, nor can they change the color context in any way. Thus, all variables are consistently colored.

$$\frac{(\chi \Rightarrow \mathit{cnstr}(\mathrm{d.f})) \quad \chi \vdash \mathrm{d}}{\chi \vdash \mathrm{d.f}} \tag{5.9}$$

When the expression preceding a field selector is consistently colored and the color environment implies the color constraint on the field, the entire field selection expression is consistently colored. This matches up with the intuitive idea that all references to a field must come from color environments that imply the color constraint on that field.

$$\frac{(\chi \Rightarrow \mathit{cnstr}(\mathrm{e.m})) \quad \chi \vdash \overline{\mathrm{d}} \quad \chi \vdash \mathrm{e}}{\chi \vdash \mathrm{e.m}\left(\overline{\mathrm{d}}\right)} \tag{5.10}$$

When the expression preceding a method invocation is consistently colored and the expressions for the actual arguments are consistently colored and the color environment implies the constraint on the method, the entire method invocation expression is consistently colored. This matches well with the intuitive idea that all method invocations must take place in a color environment that impose the color constraint on the method.

$$\frac{\chi \Rightarrow \mathit{cnstr}\left(\texttt{new}\ \mathrm{C}\left(\overline{\mathrm{d}}\right)\right) \quad \chi \vdash \overline{\mathrm{d}}}{\chi \vdash \texttt{new}\ \mathrm{C}\left(\overline{\mathrm{d}}\right)} \tag{5.11}$$

Similarly, when the color environment implies the constraint on a constructor and the expressions for the actual arguments are consistently colored, the entire constructor invocation is consistently colored. Again, this matches well with the intuition that constructor invocations must take place only in color environments that imply the color constraint on the constructor.

$$\frac{\chi \vdash \mathrm{d}}{\chi \vdash (\mathrm{C})\,\mathrm{d}} \tag{5.12}$$

[3]We use the $\Rightarrow$ operator for logical implication. Note also that the expression $(\chi \Rightarrow \mathit{cnstr}(\mathrm{e}))$ evaluates to true for all cases where $\mathit{cnstr}(\mathrm{e})$ returns $\bullet$; see Eq. (5.14). This ensures that all omitted constraints are automatically satisfied by any color environment. Further, the x in Eq. (5.8) is always either `this` or a parameter to the current method or constructor.

Casts have no effect on the color environment, and cannot have color constraints, so any cast is consistently colored when the expression being cast is itself consistently colored.

$$\frac{\Gamma \vdash \texttt{this} \in \mathrm{A} \quad \mathit{fields}(\mathrm{A}) = \overline{\mathrm{C\,f\,c}} \quad \forall \mathrm{f}_i \in \overline{\mathrm{f}}, \chi \vdash \texttt{this}.\mathrm{f}_i}{\chi \vdash \texttt{this}.\overline{\mathrm{f}} = \overline{\mathrm{f}}} \tag{5.13}$$

Field assignment in FJ and FJC is permitted only in constructors, and only for fields declared directly in the constructor's class. Furthermore, the $\overline{\mathrm{f}}$ on the right-hand side of the field assignment is restricted to be arguments from the constructor. Thus, the only possible color constraint is found on the field references on the left-hand side, all of which are to fields from the constructor's class. When each of these field references is consistently colored, all the field assignments in the constructor are consistently colored.

$$\frac{\mathit{cnstr}\,(\mathrm{e}) = \bullet}{\chi \Rightarrow \mathit{cnstr}\,(\mathrm{e})} \tag{5.14}$$

Intuitively, any color environment whatsoever implies the empty color constraint.

Example In the context of the example definitions from Figure 5.1 on page 122, evaluation of the judgment $\mathrm{c1} \wedge \mathrm{c3} \vdash \texttt{new}\,\mathrm{Pair}(\texttt{new}\,\mathrm{A}(), \texttt{new}\,\mathrm{B}()).\mathrm{setfst}(\texttt{new}\,\mathrm{B}())$ is shown below. Note that at each step we have $\boxed{\text{boxed}}$ the clauses that are being evaluated so we may proceed.

Logic	Reason
$\mathrm{c1} \wedge \mathrm{c3} \vdash \boxed{\texttt{new}\,\mathrm{Pair}(\texttt{new}\,\mathrm{A}(), \texttt{new}\,\mathrm{B}()).\mathrm{setfst}(\texttt{new}\,\mathrm{B}())}$	Example
$\boxed{((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \mathrm{c3})} \wedge \boxed{((\mathrm{c1} \wedge \mathrm{c3}) \vdash \texttt{new}\,\mathrm{B}\,())} \wedge$ $((\mathrm{c1} \wedge \mathrm{c3}) \vdash \texttt{new}\,\mathrm{Pair}\,(\texttt{new}\,\mathrm{A}\,()\,,\ \texttt{new}\,\mathrm{B}\,()))$	Boolean logic and (5.10)
$(\mathit{true}) \wedge \boxed{((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \bullet)} \wedge \boxed{(\mathrm{c1} \wedge \mathrm{c3}) \vdash \texttt{new}\,\mathrm{Pair}(\texttt{new}\,\mathrm{A}(), \texttt{new}\,\mathrm{B}())}$	(5.14) and (5.11)
$(\mathit{true}) \wedge (\mathit{true}) \wedge \left(\boxed{(\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \mathrm{c3}} \wedge \boxed{(\mathrm{c1} \wedge \mathrm{c3}) \vdash \texttt{new}\,\mathrm{A}(), \texttt{new}\,\mathrm{B}()}\right)$	Boolean logic and (5.11)
$(\mathit{true}) \wedge (\mathit{true}) \wedge (\mathit{true}) \wedge \left(\boxed{(\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \mathrm{c1}} \wedge \boxed{(\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \bullet}\right)$	Boolean logic and (5.11)
true	Boolean logic and (5.14)

So our example expression is consistently colored.

`Grant` and `Revoke`:[4] The `Grant` and `Revoke` annotations change the color bindings of the current thread. Intuitively speaking, `Grant` adds colors to a thread's color bindings, and `Revoke` removes them. To define the effect of these annotations, we must be able to "cross out" the appearances of a variable in a Boolean expression. Let P be a Boolean expression in disjunctive normal form. Consider the Boolean expression $\mathrm{P}' = [\mathrm{true}/\mathrm{x}]\,\mathrm{P}$ that is, P with true substituted for x. P' can be constructed from P by considering each conjunctive clause C_i of P, *i.e.* $\mathrm{P} = \mathrm{C}_1 \vee \cdots \vee \mathrm{C}_n$:

- If x appears positively in C_i, *i.e.*, $\mathrm{C}_i = \mathrm{x} \wedge \mathrm{D}$, then D is a conjunctive clause of P' because $\mathrm{true} \wedge \mathrm{y} = \mathrm{y}$.
- If x appears negatively in C_i, *i.e.*, $\mathrm{C}_i = \neg\mathrm{x} \wedge \mathrm{D}$, then C_i makes no contribution to P' because $\mathrm{false} \wedge \mathrm{y} = \mathrm{false}$.
- If x does not appear in C_i, then C_i is a conjunctive clause of P'.

Similarly, the Boolean expression $\mathrm{P}'' = [\mathrm{false}/\mathrm{x}]\,\mathrm{P}$ is P with all negative appearances of the variable x removed and with all *clauses* that contain positive appearances of x removed. It follows that $\mathrm{P}' \vee \mathrm{P}''$ is P with all positive and negative appearances of x removed, but otherwise unchanged. Because the range of x is

[4]Dr. Aaron Greenhouse provided significant guidance in clarifying this discussion of the `Grant` and `Revoke` operations.

$\{\mathrm{true}, \mathrm{false}\}$, $\mathrm{P}' \vee \mathrm{P}'' = \exists \mathrm{x}.\mathrm{P}$. Thus, the existential quantifier effectively removes all uses of a variable from an expression.[5]

Let us now return to `Grant`, `Revoke` and color environments. Given a color environment χ, we can create a new color environment in which all color bindings must include color cname by removing all existing appearances of cname from χ and then explicitly stating that cname must be held: $(\exists \mathrm{cname}.\chi) \wedge \mathrm{cname}$. Similarly, we can create a new color environment in which all color bindings must not include color cname: $(\exists \mathrm{cname}.\chi) \wedge \neg \mathrm{cname}$.

We need two additional helper functions to support our description of the operation of the `Grant` and `Revoke` annotations: *head* and *tail*. These functions have the intuitively obvious semantics. Each takes a possibly empty sequence as an argument; *head* returns the first item of the sequence, *tail* produces a sequence consisting of items 2..n of the sequence.

$$\frac{|\overline{\mathrm{cn}}| > 0}{\mathit{head}\,(\mathrm{cn}) = \mathrm{cn}_1} \tag{5.15}$$

$$\mathit{tail}\,(\bullet) = \bullet \tag{5.16}$$

$$\frac{|\overline{\mathrm{cn}}| = 1}{\mathit{tail}\,(\overline{\mathrm{cn}}) = \bullet} \tag{5.17}$$

$$\frac{|\overline{\mathrm{cn}}| > 0}{\mathit{tail}\,(\mathrm{cn}) = \mathrm{cn}_2, \ldots, \mathrm{cn}_{|\overline{\mathrm{cn}}|}} \tag{5.18}$$

We are now ready to define two helper functions *grant* and *revoke* that will be used when making color environment judgments:

$$\mathit{grant}\,(\chi, \overline{\mathrm{cn}}) = \begin{cases} \chi & \text{if } |\overline{\mathrm{cn}}| = 0 \\ \mathit{grant}\,(\mathit{grantone}\,(\chi, \mathit{head}\,(\overline{\mathrm{cn}}))\,, \mathit{tail}\,(\overline{\mathrm{cn}})) & \text{if } |\overline{\mathrm{cn}}| > 0 \end{cases} \tag{5.19}$$

$$\mathit{grantone}\,(\chi, \mathrm{cn}) = ((\exists \mathrm{cn}.\chi) \wedge \mathrm{cn}) \tag{5.20}$$

So *grant* is a function whose arguments are a color environment and a possibly empty list of color names resulting in a new color environment; the listed names are included in each color binding set in the new environment. Similarly, *grantone* is a function whose arguments are a color environment and a color name, and whose result is a new color environment; the color name argument is included in each color binding that is a member of the resulting color environment.

$$\mathit{revoke}\,(\chi, \overline{\mathrm{cn}}) = \begin{cases} \chi & \text{if } |\overline{\mathrm{cn}}| = 0 \\ \mathit{revoke}\,(\mathit{revokeone}\,(\chi, \mathit{head}\,(\overline{\mathrm{cn}})\,, \mathit{tail}\,(\mathrm{cn}))) & \text{if } |\overline{\mathrm{cn}}| > 0 \end{cases} \tag{5.21}$$

$$\mathit{revokeone}\,(\chi, \mathrm{cn}) = ((\exists \mathrm{cn}.\chi) \wedge \neg \mathrm{cn}) \tag{5.22}$$

So *revoke* is a function whose arguments are a color environment and a possibly empty list of color names; its result is a new color environment. The listed names are excluded from each color binding set in the new environment. Similarly, *revokeone* is a function whose arguments are a color environment and a color name, and whose result is a new color environment; the color name argument is excluded from each color binding that is a member of the resulting color environment.

So, for example

$$\begin{aligned} \mathit{grant}\,((\mathrm{c1} \wedge \mathrm{c2}) \vee (\mathrm{c1} \wedge \neg \mathrm{c3})\,,\ \mathrm{c3}) &\equiv ((\mathrm{c1} \wedge \mathrm{c2} \wedge \mathrm{c3}) \vee (\mathrm{c1} \wedge \mathrm{c3})) \\ \mathit{grant}\,((\mathrm{c1} \wedge \neg \mathrm{c2}) \vee (\neg \mathrm{c1} \wedge \mathrm{c2})\,,\ \mathrm{c3}) &\equiv ((\mathrm{c1} \wedge \neg \mathrm{c2} \wedge \mathrm{c3}) \vee (\neg \mathrm{c1} \wedge \mathrm{c2} \wedge \mathrm{c3})) \\ \mathit{grant}\,((\mathrm{c1} \wedge \neg \mathrm{c2}) \vee (\neg \mathrm{c1} \wedge \mathrm{c2})\,,\ \mathrm{c2}) &\equiv (\mathrm{c1} \wedge \mathrm{c2}) \vee (\neg \mathrm{c1} \wedge \mathrm{c2}) \\ \mathit{grant}\,((\mathrm{c1} \wedge \neg \mathrm{c2}) \vee (\neg \mathrm{c1} \wedge \mathrm{c2})\,,\ (\mathrm{c2}, \mathrm{c4})) &\equiv (\mathrm{c1} \wedge \mathrm{c2} \wedge \mathrm{c4}) \vee (\neg \mathrm{c1} \wedge \mathrm{c2} \wedge \mathrm{c4}) \end{aligned}$$

[5]Thanks to Dr. Randal Bryant for pointing out that the existential quantifier is the operator needed to “cross out” terms from a Boolean expression.

and

$$\begin{array}{rcl}
\textit{revoke}\left((\mathrm{c1} \wedge \mathrm{c2}) \vee (\mathrm{c1} \wedge \neg\mathrm{c3})\,,\ \mathrm{c3}\right) & \equiv & ((\mathrm{c1} \wedge \mathrm{c2} \wedge \neg\mathrm{c3}) \vee (\mathrm{c1} \wedge \neg\mathrm{c3})) \\
\textit{revoke}\left((\mathrm{c1} \wedge \neg\mathrm{c2}) \vee (\neg\mathrm{c1} \wedge \mathrm{c2})\,,\ \mathrm{c3}\right) & \equiv & ((\mathrm{c1} \wedge \neg\mathrm{c2} \wedge \neg\mathrm{c3}) \vee (\neg\mathrm{c1} \wedge \mathrm{c2} \wedge \neg\mathrm{c3})) \\
\textit{revoke}\left((\mathrm{c1} \wedge \neg\mathrm{c2}) \vee (\neg\mathrm{c1} \wedge \mathrm{c2})\,,\ \mathrm{c2}\right) & \equiv & (\mathrm{c1} \wedge \neg\mathrm{c2}) \vee (\neg\mathrm{c1} \wedge \neg\mathrm{c2}) \\
\textit{revoke}\left((\mathrm{c1} \wedge \neg\mathrm{c2}) \vee (\neg\mathrm{c1} \wedge \mathrm{c2})\,,\ (\mathrm{c2}, \mathrm{c4})\right) & \equiv & (\mathrm{c1} \wedge \neg\mathrm{c2} \wedge \neg\mathrm{c4}) \vee (\neg\mathrm{c1} \wedge \neg\mathrm{c2} \wedge \neg\mathrm{c4})
\end{array}$$

where $\equiv$ denotes logical equivalence.

Method overriding auxiliary definitions: Intuitively speaking, it seems clear that color constraints should be considered in judging the validity of method overriding. To make this judgment we need some additional definitions. When a method $\mathrm{m_c}$—the "child" method—legally overrides another method $\mathrm{m_p}$—the "parent" method—in FJ (that is, the override is permitted according to FJ's type system), we say that $\textit{overrideFJ}\,(\mathrm{m_c}, \mathrm{m_p})$ is true; thus $\textit{overrideFJ}$ is a function from two methods to a boolean result. We also define $\textit{overrides}\,(\mathrm{m_p})$ to be the set of all methods in the program that override method $\mathrm{m_p}$. We use these definitions to define a function $\textit{overrideCol}$, taking two methods as arguments, and returning a boolean. Specifically,

$$\textit{overrideCol}\,(\mathrm{m_c}, \mathrm{m_p}) = \textit{overrideFJ}\,(\mathrm{m_c},\ \mathrm{m_p}) \wedge (\textit{cnstr}\,(\mathrm{m_p}) \Rightarrow \textit{cnstr}\,(\mathrm{m_c})) \tag{5.23}$$

So $\textit{overrideCol}$ is true only when both method $\mathrm{m_c}$ legally overrides method $\mathrm{m_p}$ and the constraint on method $\mathrm{m_p}$ implies the constraint on $\mathrm{m_c}$. We will use $\textit{overrideCol}$ in our judgment of the coloring of methods, below.

$$\textit{overrides}\,(\mathrm{m_p}) = \{\mathrm{m} \in \mathrm{Cls}\,|\,\textit{overrideFJ}\,(\mathrm{m},\ \mathrm{m_p})\} \tag{5.24}$$

Color environment judgments

We use the judgment

$$\mathrm{Context} \models \mathrm{FJCfragment}$$

to denote consistent coloring for methods, constructors, classes and programs. The context is drawn from the implicitly present fixed program. Thus, when Cls appears on the left hand side, we are depending on program text to allow the judgment; when Col appears we are depending either on the universe of color names, on the globally scoped color annotations (see Section 5.3.2 on page 130 for details on these), or on both. Color environment judgments are read as "in the context of the program, the given FJC fragment is consistently colored."

To ease the introduction of later expansions to the coloring model, we introduce two helper functions for granting and revoking colors: $\textit{grantx}$ and $\textit{revokex}$, defined as follows:

$$\textit{grantx}\,(\chi, \overline{\mathrm{cn}}) = \textit{grant}\,(\chi, \overline{\mathrm{cn}}) \tag{5.25}$$

$$\textit{revokex}\,(\chi, \overline{\mathrm{cn}}) = \textit{revoke}\,(\chi, \overline{\mathrm{cn}}) \tag{5.26}$$

These functions have the same domain and range as *grant* and *revoke*; their current "pass-through" definitions are replaced in a later extension to the coloring model.

For methods: For methods that do not override another method, our only concern is whether the method body is consistently colored.

$$\frac{\textit{grantx}(\textit{revokex}(\mathrm{c}, \overline{\mathrm{r}}),\ \overline{\mathrm{a}}) \vdash \mathrm{e}_0 \quad \forall \mathrm{m1} \in \mathrm{Cls},\ \mathrm{m} \notin \textit{overrides}\,(\mathrm{m1})}{\mathrm{Cls} \models \mathrm{C}_0\ \mathrm{m}\left(\overline{\mathrm{C\,x}}\right)\ \texttt{Color}\ \mathrm{c}\ \{\texttt{Revoke}\ \overline{\mathrm{r}}\,;\ \texttt{Grant}\ \overline{\mathrm{a}}\,;\ \texttt{return}\ \mathrm{e}_0\,;\}} \tag{5.27}$$

The conclusion of this equation should be read "in the current program, method m is consistently colored." Likewise, the premises should be read as "in the color environment created by applying first the `Revokes` and then the `Grants` to c, e_0's constraints are satisfied," and "method m does not override any other method."

Similarly, for methods that *do* override some other method, we require that the child's color constraint be consistent with the parent method's constraint.

$$\frac{\mathit{grantx}(\mathit{revokex}(\mathrm{c}, \bar{\mathrm{r}}),\, \bar{\mathrm{a}}) \vdash \mathrm{e}_0 \quad \mathit{overrideCol}\,(\mathrm{m},\, \mathrm{m_p})}{\mathrm{Cls} \models \mathrm{C}_0\ \mathrm{m}\,\left(\overline{\mathrm{C\,x}}\right)\ \texttt{Color}\ \mathrm{c}\ \{\texttt{Revoke}\ \bar{\mathrm{r}};\ \texttt{Grant}\ \bar{\mathrm{a}};\ \texttt{return}\ \mathrm{e}_0;\}} \tag{5.28}$$

The premises should be read as "in the color environment created by applying first the `Revoke`s and then the `Grant`s to c, e_0's constraints are satisfied," and "method m overrides method $\mathrm{m_p}$ and $\mathrm{m_p}$'s constraint implies m's constraint." Note that this rule requires that all overriding methods have constraints that are no stronger than that found on the method they override. This property maintains substitutability.[6] We'll relax this requirement later.

We now work through the color judgment for method setfst from our running example (see Figure 5.1 on page 122). As before, the portion to be evaluated next is boxed.

```
Pair setfst(Object newfst) color c3 {
    return new Pair(newfst, this.snd);
  }
```

Logic	Reason
$\boxed{\mathit{grant}\,(\mathit{revoke}\,(\mathrm{c3}, \bullet)\,, \bullet)} \vdash \texttt{new}\ \mathrm{Pair}\,(\mathrm{newfst},\ \texttt{this}.\mathrm{snd})$	premise of (5.27)
$\boxed{\mathrm{c3} \vdash \texttt{new}\ \mathrm{Pair}\,(\mathrm{newfst},\ \texttt{this}.\mathrm{snd})}$	(5.19)
$\left(\boxed{(\mathrm{c3} \Rightarrow \mathit{cnstr}\,(\texttt{new}\ \mathrm{Pair}))}\right) \wedge (\mathrm{c3} \vdash \mathrm{newfst},\ \texttt{this}.\mathrm{snd})$	(5.11)
$(\mathrm{c3} \Rightarrow \mathrm{c3}) \wedge \left(\boxed{\mathrm{c3} \vdash \mathrm{newfst},\ \texttt{this}.\mathrm{snd}}\right)$	(5.7)
$(\mathrm{c3} \Rightarrow \mathrm{c3}) \wedge \Big((\mathrm{c3} \Rightarrow \bullet) \wedge \Big(\mathrm{c3} \Rightarrow \boxed{\mathit{cnstr}\,(\mathrm{Pair.snd})}\Big) \wedge (\mathrm{c3} \vdash \texttt{this})\Big)$	(5.8), (5.9)
$(\mathrm{c3} \Rightarrow \mathrm{c3}) \wedge \Big((\mathrm{c3} \Rightarrow \bullet) \wedge (\mathrm{c3} \Rightarrow \bullet) \wedge \Big(\boxed{\mathrm{c3} \vdash \texttt{this}}\Big)\Big)$	(5.5)
$\left(\boxed{(\mathrm{c3} \Rightarrow \mathrm{c3}) \wedge ((\mathrm{c3} \Rightarrow \bullet) \wedge (\mathrm{c3} \Rightarrow \bullet) \wedge (\mathit{true}))}\right)$	(5.8)
true	Boolean Logic, (5.14)

Because the premise is true, the judgment is that method setfst is consistently colored.

For constructors: The judgment for constructors uses a new piece of notation. We write $\chi \vdash \{\bar{\mathrm{e}}\}$ to abbreviate the color judgments $\chi \vdash \mathrm{e}_1, \ldots \chi \vdash \mathrm{e}_n$.

$$\frac{\mathit{grant}(\mathit{revoke}(\mathrm{c}, \bar{\mathrm{r}}),\, \bar{\mathrm{a}}) \vdash \left\{\mathrm{super}\,(\bar{\mathrm{g}}),\ \texttt{this}.\bar{\mathrm{f}} = \bar{\mathrm{f}}\right\}}{\mathrm{Cls} \models \mathrm{C}\,\left(\overline{\mathrm{D\,g}},\ \overline{\mathrm{C\,f}}\right)\ \mathrm{c}\ \left\{\bar{\mathrm{r}};\ \bar{\mathrm{a}};\ \mathrm{super}\,(\bar{\mathrm{g}})\ ;\ \texttt{this}.\bar{\mathrm{f}} = \bar{\mathrm{f}}\,;\right\}} \tag{5.29}$$

This says that if the color environment created by applying first the `Revoke`s and then the `Grant`s found in the constructor to c satisfies the constraint on the constructor of **super** and also any constraints on the fields of f, the constructor is consistently colored in the current program. Returning to the code from Figure 5.1 on page 122, and using the constructor for class Pair as our example, we have:

```
class Pair extends Object {
  Object fst Color c3;
  Object snd;
  Pair(Object fst, Object snd) Color c3 {
    grant c1; super(); this.fst=fst; this.snd=snd;
```

[6]The requirement is actually too strong for real-world use. There are plenty of real examples where substitutability *cannot be* guaranteed; `Runnable.run()`, for example.

```
        }
    }
```

Logic	Reason
$\boxed{\mathit{grant}\left(\mathit{revoke}\left(\mathrm{c3}, \bullet\right), \mathrm{c1}\right)} \vdash \{\texttt{new}\ \mathrm{Object}\,(), \texttt{this}.\mathrm{fst} = \mathrm{fst}, \texttt{this}.\mathrm{snd} = \mathrm{snd}\}$	premise of (5.29)
$(\mathrm{c1} \wedge \mathrm{c3}) \vdash \left\{\boxed{\texttt{new}\ \mathrm{Object}\,()}, \texttt{this}.\mathrm{fst} = \mathrm{fst}, \texttt{this}.\mathrm{snd} = \mathrm{snd}\right\}$	(5.21), (5.19)
$(\mathrm{c1} \wedge \mathrm{c3}) \vdash \{\texttt{this}.\mathrm{fst} = \mathrm{fst}, \texttt{this}.\mathrm{snd} = \mathrm{snd}\} \wedge \Big((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \boxed{\mathit{cnstr}\,(\texttt{new}\ \mathrm{Object}())} \wedge \boxed{((\mathrm{c1} \wedge \mathrm{c3}) \vdash \bullet)}\Big)$	(5.11)
$(\mathrm{c1} \wedge \mathrm{c3}) \vdash \left\{\boxed{\texttt{this}.\mathrm{fst} = \mathrm{fst}}, \boxed{\texttt{this}.\mathrm{snd} = \mathrm{snd}}\right\} \wedge (((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \bullet) \wedge \mathit{true})$	(5.7),(5.14)
$\Big(((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \mathit{cnstr}\,(\mathrm{Pair.fst})) \wedge \boxed{((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \mathit{cnstr}\,(\texttt{this}))}\Big) \wedge \Big(((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \mathit{cnstr}\,(\mathrm{Pair.snd})) \wedge \boxed{((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \mathit{cnstr}\,(\texttt{this}))}\Big) \wedge \Big(\boxed{((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \bullet)} \wedge \mathit{true}\Big)$	(5.13)
$\Big(\Big((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \boxed{\mathit{cnstr}\,(\mathrm{Pair.fst})}\Big) \wedge \mathit{true}\Big) \wedge \Big(\Big((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \boxed{\mathit{cnstr}\,(\mathrm{Pair.snd})}\Big) \wedge \mathit{true}\Big) \wedge (\mathit{true} \wedge \mathit{true})$	(5.8), (5.14)
$(((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \mathrm{c3}) \wedge \mathit{true}) \wedge \Big(\boxed{((\mathrm{c1} \wedge \mathrm{c3}) \Rightarrow \bullet)} \wedge \mathit{true}\Big) \wedge (\mathit{true} \wedge \mathit{true})$	(5.5)
true	Boolean logic and (5.14)

Because the premise of (5.29) is true, the constructor new Pair() is consistently colored.

For classes: A class is consistently colored when all its methods and its constructor are consistently colored.

$$\frac{\mathrm{Cls} \models \mathrm{K} \quad \mathrm{Cls} \models \overline{\mathrm{M}}}{\mathrm{Cls} \models \texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{D}\ \left\{\overline{\mathrm{C\ f\ c}};\ \mathrm{K}\,\overline{\mathrm{M}}\right\}} \tag{5.30}$$

For programs: A program is consistently colored if all the classes in the universe of classes are consistently colored.

$$\frac{\forall \mathrm{C} \in \mathrm{Cls}, \mathrm{Cls} \models \mathrm{C}}{\mathrm{Cls} \models \mathrm{Cls}} \tag{5.31}$$

5.3 Expanded Model

5.3.1 Method overriding

In actual Java programs we often see overriding methods where the parent and child methods do not have the same color constraint. This is not a problem when the child method's constraint is implied by the parent method's constraint. Overridings where the child's constraint is incompatible with the parent's constraint are problematic, however. `Runnable.run()` is a prime example—the parent method has no constraint, yet the children may have arbitrary constraints. Consider, for example, `Runnable`s being handed to `SwingUtilities.invokeLater(Runnable)`. The `run` method of these `Runnable`s will certainly be invoked on the `AWT` event thread. In typical usage, these `Runnable`s often call methods that must be invoked *only* on the event thread.

Valid method overriding:

In order to support more flexible method overriding, we add a rule for $overrideCol$ as follows:

$$\frac{\begin{array}{c} \mathit{overrideFJ}\left(\mathrm{m_c},\, \mathrm{m_p}\right) \quad \mathit{cnstr}\left(\mathrm{m_p}\right) \not\Rightarrow \mathit{cnstr}\left(\mathrm{m_c}\right) \\ \mathit{incompatible\ override\ warning} \end{array}}{\mathit{overrideCol}\left(\mathrm{m_c}, \mathrm{m_p}\right) = \mathit{true}} \tag{5.32}$$

This additional rule for method overriding permits overrides whose constraint is incompatible with that of the overridden method; the premise $\mathit{incompatible\ override\ warning}$ represents the presence of an analysis-time warning of an incompatible method overriding. Once again we use $\mathrm{m_c}$ to denote some child method and $\mathrm{m_p}$ to denote some parent method. While necessary for thread coloring of real-world programs, it opens the possibility of unexpected behavior on method invocation. With *only* this rule added to those above, we could guarantee correct behavior only for programs without an incompatible-override warning. This would be an unacceptable gap in the coloring system.

Method invocation:

To partially close the gap, we modify the coloring judgment for method invocation. The goal is to require that any specific method invocation happens in a color environment that satisfies the constraints of all possible overridings of the named method. This places us back on a sound footing. The over-ridings of, *e.g.* `Runnable.run()` may not all have consistent constraints. But the program remains consistent as long as the current color context at any individual call site satisfies the constraints of all of the methods potentially invoked from that particular call site.

$$\frac{\chi \Rightarrow \mathit{cnstr}(\mathrm{m}) \quad \forall m' \in \mathit{overrides}(m),\, \chi \vdash m' \quad \chi \vdash \overline{\mathrm{d}} \quad \chi \vdash \mathrm{e}}{\chi \vdash \mathrm{e.m}\left(\overline{\mathrm{d}}\right)} \tag{5.33}$$

We're changing only the method invocation judgment, all other cases remain unchanged.

While this loosening of rules for method over-riding makes it easier to handle some methods like `Runnable.run()`, we really need a more powerful approach for full generality. In particular, this modification cannot help us when the conjunction of the constraints for the overriding methods is unsatisfiable, such as `Runnable.run()`. For such cases, we will need an extension to support parameterized colors. See Section 5.5.2 for a description of some attributes of the "real" solution.

5.3.2 Incompatible colors

Many patterns of color usage depend on the idea that certain colors are never simultaneously associated with any one thread. We express this with the `IncompatibleColors` annotation. The advantage given by incompatible colors is that knowing that a color binding contains one of them is sufficient to know that it *does not* contain any of the others.

Syntax

$$ic \quad ::= \quad \texttt{IncompatibleColors}\, \overline{\mathit{cnames}}$$

An `IncompatibleColors` statement may appear anywhere a class definition would have appeared in our program. There may be more than one such statement. We assume that these statements pass a basic sanity check: no color may be declared to be incompatible with itself, nor may any `IncompatibleColors` statement name a color that does not appear in the color universe Col.

Semantics

In this section we define the changes required to support incompatible colors. We must answer three questions:

1. Given the possibility of incompatible colors, under what conditions does a color environment satisfy a `Grant` or `Revoke` annotation? This was not an issue in the simpler version of our semantics, as without incompatible colors it is impossible for a `Grant` to be invalid. Answering this question does not require knowing what effect the annotation has on the color environment.
2. What happens to the color environment when you `Grant` or `Revoke` a color that is incompatible with another (or some other) colors? In (5.19) through (5.22) on page 126 we defined the effect of `Grant` and `Revoke` in the absence of incompatibilities.
3. Given the possibility of incompatible colors, when is a method or constructor consistently colored? This judgment requires the answers to the first two questions.

Helper functions As usual in DNF, we refer to variables and their negations as *literals*. For example, x_1 and $\neg x_2$ are both literals; the first is a *positive* literal and the second a *negative* literal. The expression produced when we use the AND operator $\wedge$ over a group of literals is a *clause*, such as $(x_1 \wedge \neg x_2)$.

To define the effect of the `IncompatibleColors` statement, we first define three helper functions:

- The *xflict* function computes a Boolean expression over color names that represents the effect of a single `IncompatibleColors` statement on the given color.

$$\frac{\texttt{IncompatibleColors}\ \overline{\mathrm{cn}} \quad \mathrm{cn}_i \in \overline{\mathrm{cn}}}{\mathit{xflict}\,(\overline{\mathrm{cn}}, \mathrm{cn}_i) = (\bigotimes \overline{\mathrm{cn}})} \tag{5.34}$$

$$\frac{\texttt{IncompatibleColors}\ \overline{\mathrm{cn}} \quad \mathrm{cname} \notin \overline{\mathrm{cn}}}{\mathit{xflict}\,(\overline{\mathrm{cn}}, \mathrm{cname}) = \mathit{true}} \tag{5.35}$$

  We use $\bigotimes$ as the multi-way exclusive-or operation, that is

$$\otimes\,\overline{\mathrm{cn}} = \bigvee_{i=1}^{n}\left(\bigwedge_{k=1}^{n}\left(\begin{cases} i = k & \mathrm{cn}_k \\ i \neq k & \neg\mathrm{cn}_k \end{cases}\right)\right) \vee \left(\bigwedge_{i=1}^{n} \neg\mathrm{cn}_i\right) \tag{5.36}$$

  Thus, $\mathit{xflict}\,(\overline{cn}, cn_i)$ results in *at most one* of the named colors. Note that Eq. (5.35) ensures that conflicts have no effect when either the named color is not declared to be incompatible with any other color, or when *no* colors are declared to be incompatible.

- The *conflict* function produces a Boolean expression over color names. The expression represents the combined effect on the given color of *every* `IncompatibleColors` statement in the entire program.

$$\frac{\texttt{IncompatibleColors}\ \overline{\mathrm{cn}_1} \ldots \texttt{IncompatibleColors}\ \overline{\mathrm{cn}_x}}{\mathit{conflict}\,(\mathrm{cname}) = \bigwedge_{i=1}^{x} \mathit{xflict}\,(\overline{\mathrm{cn}_i}, \mathrm{cname})} \tag{5.37}$$

- The *poslits* function selects exactly and only the positive literals from a DNF color expression, returning them as a set that may be empty.

$$\mathit{poslits}\,(\mathit{cExp}) = \{\mathit{all\ positive\ literals\ in\ cExp}\} \tag{5.38}$$

As an additional example, given the declarations

```
IncompatibleColors a, b
IncompatibleColors a, c
```

the above functions evaluate as:

Function	Value
$\mathit{xflict}\,(\langle \mathrm{a}, \mathrm{b}\rangle, \mathrm{a})$	$((\mathrm{a} \wedge \neg \mathrm{b}) \vee (\neg \mathrm{a} \wedge \mathrm{b})) \vee (\neg \mathrm{a} \wedge \neg \mathrm{b})$
$\mathit{xflict}\,(\langle \mathrm{a}, \mathrm{c}\rangle, \mathrm{a})$	$((\mathrm{a} \wedge \neg \mathrm{c}) \vee (\neg \mathrm{a} \wedge \mathrm{c})) \vee (\neg \mathrm{a} \wedge \neg \mathrm{c})$
$\mathit{xflict}\,(\langle \mathrm{a}, \mathrm{c}\rangle, \mathrm{b})$	true
$\mathit{conflict}\,(\mathrm{a})$	$\left(\begin{array}{c} (\mathrm{a} \wedge \neg \mathrm{b} \wedge \neg \mathrm{c}) \vee (\neg \mathrm{a} \wedge \mathrm{b} \wedge \mathrm{c}) \vee \\ (\neg \mathrm{a} \wedge \neg \mathrm{b} \wedge \mathrm{c}) \vee (\neg \mathrm{a} \wedge \mathrm{b} \wedge \neg \mathrm{c}) \vee (\neg \mathrm{a} \wedge \neg \mathrm{b} \wedge \neg \mathrm{c}) \end{array}\right)$
$\mathit{conflict}\,(\mathrm{b})$	$((\mathrm{a} \wedge \neg \mathrm{b}) \vee (\neg \mathrm{a} \wedge \mathrm{b})) \vee (\neg \mathrm{a} \wedge \neg \mathrm{b})$

Satisfaction of `Grant` annotation To add a color to a color environment, we must be be able to demonstrate that none of the colors that conflict with it are present in any of the color bindings that might be active in the color environment prior to the `Grant`. As we have seen, $\mathit{conflict}\,(\mathrm{cname})$ is the Boolean expression that describes the complete constraint for color cname. Because $\wedge$ distributes over $\vee$ and $x \wedge \neg x = \mathrm{false}$, $\mathrm{cname} \wedge \mathit{conflict}\,(\mathrm{cname})$ is $\mathit{conflict}\,(\mathrm{cname})$ with all the clauses that contain negative uses of cname removed. That is, it describes all the color bindings in which cname is held. Eliminating cname from the expression entirely, $\exists \mathrm{cname}.\mathrm{cname} \wedge \mathit{conflict}\,(\mathrm{cname})$, yields the precondition that the color environment must satisfy before adding cname to all the color bindings in the environment.

$$\frac{|\overline{\mathrm{cn}}| > 0 \quad \bigwedge_{i=1}^{|\overline{\mathrm{cn}}|} (\chi \Rightarrow (\exists \mathrm{cn}_i. (\mathrm{cn}_i \wedge \mathit{conflict}\,(\mathrm{cn}_i))))}{\chi \vdash \texttt{Grant}\, \overline{\mathrm{cn}}} \tag{5.39}$$

$$\frac{}{\chi \vdash \texttt{Grant}\, \bullet} \tag{5.40}$$

Note that because `Revoke` only removes colors from the context and does not add colors to the context, it can never cause a color error itself, even though it may cause a later color consistency judgment to fail.

$$\frac{}{\chi \vdash \texttt{Revoke}\, \overline{\mathrm{cn}}} \tag{5.41}$$

We omit revoke from all color satisfaction judgments for this reason.

For example, in the context of the code fragment

```
    IncompatibleColors a, b
```

the evaluation of $\mathrm{a} \vdash \texttt{Grant}\mathrm{b}$ is simply the evaluation of $(\mathrm{a} \Rightarrow \exists \mathrm{b}. (\mathrm{b} \wedge ((\mathrm{a} \wedge \neg \mathrm{b}) \vee (\neg \mathrm{a} \wedge \mathrm{b}) \vee (\neg \mathrm{a} \wedge \neg \mathrm{b}))))$, which simplifies to $(\mathrm{a} \Rightarrow \neg \mathrm{a})$ which is, of course, false. So granting b in the context of a is not permitted when a and b are incompatible. This is exactly the behavior we wanted.

Definition of `Grant` and `Revoke` with conflicts When incompatible colors are supported, the definition of grantx and $\mathit{revokex}$ must also change. In particular, the changes these functions make to the color environment must correctly reflect the incompatibilities. So Eq. (5.25) changes to

$$\frac{\mathit{grant}\,(\chi, \overline{\mathrm{cn}}) = \chi' \quad |\overline{\mathrm{cn}}| > 0}{\mathit{grantx}\,(\chi, \overline{cn}) = [\forall \mathit{cname} \in \mathit{poslits}\,(\chi'), ((\mathit{cname} \wedge \mathit{conflict}\,(\mathit{cname}))\,/\,\mathit{cname}_{pos})]\,\chi'} \tag{5.42}$$

and Eq. (5.26) changes to

$$\frac{\mathit{revoke}\,(\chi, \overline{\mathrm{cn}}) = \chi' \quad |\overline{\mathrm{cn}}| > 0}{\mathit{revokex}\,(\chi, \overline{cn}) = [\forall \mathit{cname} \in \mathit{poslits}\,(\chi'), ((\mathit{cname} \wedge \mathit{conflict}\,(\mathit{cname}))\,/\,\mathit{cname}_{pos})]\,\chi'} \tag{5.43}$$

$$\mathit{grantx}\,(\chi, \bullet) = \mathit{revokex}\,(\chi, \bullet) = \chi \tag{5.44}$$

We use the operation $/cname_{pos}$ to indicate substitution only for the positive literals of $cname$, and not for any negative literals. Our need for the operation $/cname_{pos}$ becomes clear when we consider the problem of correctly maintaining the global constraints implied by the `IncompatibleColors` statements in the program. When a color binding includes a color that has conflicts, it must be the case that the color binding does not include any of the conflicting colors. We enforce this property by substituting the colorname AND-d with its conflicts for the positive reference to the color name, as in $(cname \wedge conflict\,(cname))\,/cname_{pos}$. There is, of course, no need to substitute for negative mention of the colorname, because lack of a color does not bring with it the global "at most one" constraint. As with Eqs. (5.19) and (5.21), empty $grantx$ and $revokex$ have no effect.

A reasonable alternate definition of `Grant` in the presence of incompatible colors would use "make it so" semantics. That is, `Grant`ing a color would automatically `Revoke` any color with which the granted color is in conflict. We avoided this alternative for pragmatic reasons—it would be too easy to "lie" in the code, by simply `Grant`ing the color you need and pressing forward. An example might be `Grant`ing the `AWT` color while executing in a `Compute` thread. The definition presented above is a better match for typical usage. It also has the benefit that "lying" to the analysis requires two annotations—one must first explicitly `Revoke` the current color before `Grant`ing the new one. Perhaps the extra work may lead some careless user to think twice.

Constraints with `IncompatibleColors` To support incompatible colors fully, the declared incompatibilities must affect color constraints as well as `Grant`s. To this end, we modify the $cnstr$ helper function as follows:

$$\frac{\texttt{IncompatibleColors}\ \overline{incs}}{cnstr_{inc}\,(e) = [\forall cname \in poslits\,(cnstr\,(e))\,,((cname \wedge conflict\,(cname))\,/cname_{pos})]\ cnstr\,(e)} \tag{5.45}$$

To judge color constraint satisfaction and consistent color usage in an FJC program with incompatible colors, simply modify all previous definitions of the color consistency judgments for expressions—Eqs. (5.8) through (5.13)—and for methods Eq. (5.27), constructors Eq. (5.29) to use $cnstr_{inc}$ in place of the original $cnstr$, with no other changes.

Color judgments with `IncompatibleColors` We now modify Eq. (5.27) to support incompatibleColors. The change is simple: replace *grant* with *grantx* and *revoke* with *revokex* (and remember that we now depend on global color annotations in addition to the program), as follows:

$$\frac{revokex\,(\mathrm{c},\bar{\mathrm{r}}) \vdash \texttt{Grant}\ \bar{\mathrm{a}} \quad grantx\,(revokex\,(\mathrm{c},\bar{\mathrm{r}})\,,\bar{\mathrm{a}}) \vdash \mathrm{e}}{\mathrm{Col,\ Cls} \models \mathrm{C}_0\ \mathrm{m}\left(\overline{\mathrm{C\,x}}\right)\ \mathrm{c}\ \{\bar{\mathrm{r}}\,;\ \bar{\mathrm{a}};\ \texttt{return}\,\mathrm{e}\,;\}} \tag{5.46}$$

The judgment for constructors, Eq. (5.29), is modified similarly:

$$\frac{revokex\,(\mathrm{c},\bar{\mathrm{r}}) \vdash \texttt{Grant}\ \bar{\mathrm{a}} \quad grantx(revokex(\mathrm{c},\bar{\mathrm{r}}),\ \bar{\mathrm{a}}) \vdash \left\{\mathrm{super}\,(\bar{\mathrm{g}}),\ \texttt{this}.\bar{\mathrm{f}} = \bar{\mathrm{f}},\ \bar{\mathrm{g}}\right\}}{\mathrm{Cls} \models \mathrm{C}\left(\overline{\mathrm{D\,g}},\ \overline{\mathrm{C\,f}}\right)\ \mathrm{c}\ \left\{\bar{\mathrm{r}}\,;\ \bar{\mathrm{a}};\ \mathrm{super}\,(\bar{\mathrm{g}})\ ;\ \texttt{this}.\bar{\mathrm{f}} = \bar{\mathrm{f}}\,;\right\}} \tag{5.47}$$

Now suppose that we modify our running example (see Figure 5.1 on page 122) to include the declaration

```
IncompatibleColors c1, c2
```

and change the definition of `Pair.snd` to be

```
Object snd Color c2;
```

and then evaluate the color judgment for the constructor of `Pair`.

Logic	Reason
$\mathit{revokex}\,(\mathrm{c3}, \bullet) \vdash \texttt{Grant}\ c1 \quad \mathit{grantx}\,(\mathit{revoke}\,(\mathrm{c3}, \bullet), \mathrm{c1}) \vdash$ $\left\{\boxed{\texttt{new}\ \mathrm{Object}}, \mathrm{Pair.fst} = \mathrm{fst}, \mathrm{Pair.snd} = \mathrm{snd}, \boxed{\mathit{fst}, \mathit{snd}}\right\}$	premise of modified (5.29)
$\boxed{\mathit{revokex}\,(\mathrm{c3}, \bullet)} \vdash \texttt{Grant}\ \mathrm{c1} \quad \boxed{\mathit{grantx}\,(\mathit{revoke}\,(\mathrm{c3}, \bullet), \mathrm{c1})} \vdash$ $\{\mathrm{Pair.fst} = \mathrm{fst}; \mathrm{Pair.snd} = \mathrm{snd}\}$	`new Object()` has no constraint and no arguments so true,(5.8)
$\boxed{\mathrm{c3} \vdash \texttt{Grant}\ \mathrm{c1}}$ $(\mathrm{c1} \wedge \neg\mathrm{c2} \wedge \mathrm{c3}) \vdash \{\mathrm{Pair.fst} = \mathrm{fst}; \mathrm{Pair.snd} = \mathrm{snd}\}$	(5.44), (5.42)
$\mathrm{c3} \Rightarrow \left(\exists \mathrm{c1}.\left(\boxed{\mathrm{c1} \wedge \mathit{conflict}\,(\mathrm{c1})}\right)\right)$ $(\mathrm{c1} \wedge \neg\mathrm{c2} \wedge \mathrm{c3}) \vdash \{\mathrm{Pair.fst} = \mathrm{fst}; \mathrm{Pair.snd} = \mathrm{snd}\}$	(5.39)
$\mathrm{c3} \Rightarrow \boxed{(\exists \mathrm{c1}.\,(\mathrm{c1} \wedge \neg\mathrm{c2}))}$ $(\mathrm{c1} \wedge \neg\mathrm{c2} \wedge \mathrm{c3}) \vdash \{\mathrm{Pair.fst} = \mathrm{fst}; \mathrm{Pair.snd} = \mathrm{snd}\}$	(5.34)
$\mathrm{c3} \Rightarrow \neg\mathrm{c2}$ $(\mathrm{c1} \wedge \neg\mathrm{c2} \wedge \mathrm{c3}) \vdash \{\mathrm{Pair.fst} = \mathrm{fst}; \mathrm{Pair.snd} = \mathrm{snd}\}$	Boolean Logic

At this point we reach a contradiction, as $\mathrm{c3} \Rightarrow \neg\mathrm{c2}$ is clearly false. So with the addition of the `IncompatibleColors` our example program becomes inconsistent. Further, if we had evaluated the coloring judgments in a different order, we would have found that the assignment of `Pair.snd` was also inconsistent because a color environment of c1 excludes the possibility of c2.

The color consistency judgment shown here is quite specific to incompatible colors because that is what the implementation supports. There is no fundamental reasoning behind this choice; supporting general global constraints would be straight-forward both in terms of implementation and definition. However, we have not yet found a use-case for a more general global Boolean constraint.

5.3.3 Constraints on multiplicity

Some important thread usage patterns depend on the knowledge that there is at most one thread at a time that takes on a particular role. An important example is typical GUI programming in Java, where there is a single event-handling thread at any given time. This thread is the only one permitted to handle user events or to interact with the display and the GUI objects thereon. The standard AWT and Swing frameworks follow this pattern, as does the SWT windowing toolkit. The apparent motivations for this pattern are improved GUI performance and avoidance of potential deadlock. Specifically, the GUI implementers avoid locking internal state; instead, they depend on this single-event-thread discipline for state consistency.

We model this property to assist with reasoning about thread confinement. We do not, however, check the correctness of programmer promises about thread multiplicity. These promises remain as unsatisfied proof obligations.

Syntax

$$\mathit{card} \quad ::= \quad \mathrm{MaxColorCount}\ \mathit{cn}\ 1 \tag{5.48}$$

$$\mathit{card} \quad ::= \quad \mathrm{MaxColorCount}\ \mathit{cn}\ \mathrm{n} \tag{5.49}$$

The `MaxColorCount` annotation asserts a particular multiplicity for the relationship between the named color cn and the number of threads that may be associated with that color at any given moment.

Semantics

The first form of the promise (5.48) claims that there will never be more than one thread associated with cn at a time. This form is used to model the `AWT`, `Swing`, and `SWT` event threads. The second form (5.49) asserts that we do not know how many threads may be associated with cn at any time; this is semantically equivalent to remaining silent on the question. This form is nonetheless useful as documentation: it indicates that the programmer has considered the question of multiplicity and has decided that it is not known.

In FJC we assume that all color multiplicities are explicitly stated, and that the color multiplicity promises are part of the global color annotations Col.

Color multiplicity promises are declarative, and are not checked in any way.

5.4 Differences between FJC and Java

FJ—and thus FJC—deliberately omits many features of the complete Java language in order to reduce the size and effort of expression of definitions and proofs. FJ omitted any language feature that made their proof of type soundness longer without making it significantly different. For FJC we also omitted an important feature of the Fluid system: data regions. In this section we briefly discuss the impact of the missing features of FJ and FJC on both the thread coloring language and the thread coloring analysis.

Assignment Thread coloring in the absence of parameterized colors need not consider assignment at all. We must check for any color constraints on field references in both the left-hand side and right-hand side of the assignment, of course, but the assignment itself adds no additional complication. When parameterized colors are added, we'll have to check that the color parameter of the right-hand side expression matches (or is compatible with) the parameterized color type of the left-hand side. See Section 5.5.2 on page 137 for a brief discussion of some aspects of parameterized colors.

Method overloading Thread coloring need not consider method overloading at all. We simply check any constraints on the fully-bound method exactly as shown for an ordinary method invocation.

Interfaces Implementation of a Java `Interface` requires that the methods in the implementing class match those in the `interface` in all expressed particulars. Thread coloring treats methods that implement an `interface` similarly to methods that override those from parent classes. In particular, we 'inherit' color constraints from the `interface`'s method declaration and then apply exactly the same coloring consistency rules as for method overriding.

Null pointers and base types have no direct impact on thread coloring. We must, of course, check field references for color constraints in the ordinary fashion regardless of the type of the field. Similarly, we must check color constraints in expressions whether or not some reference evaluates to `null`.

Abstract classes and methods We treat abstract classes and methods as though they were ordinary methods. This means that thread coloring analysis for overridden abstract methods proceeds exactly as for any other overridden method, with the same checks on color consistency.

Shadowing of fields has no impact on thread coloring. We simply allow the binder to sort out which field we're actually referencing, and proceed as usual for references to a field.

Access control has no direct impact on thread coloring. Since we only perform thread coloring analysis on programs that compile successfully, all references we process already obey any access control restrictions. More broadly, access control does effect the inference algorithm used to reduce the number of annotations written by users. In particular, items visible between inference chunks are treated as cut points, and must be annotated sufficiently to allow the inference to proceed. This does not, however, have any impact on our analysis of color consistency.

Nested blocks FJ omits nested lexical blocks. Thread coloring permits `Grant` and `Revoke` annotations at the beginnings of blocks. We handle blocks with such annotations by pushing the current color context onto a stack at the beginning of the block, and then creating a new context by applying the annotations exactly as for method bodies in FJC. At the end of the block, we restore the color context from the stack. This poses no additional analysis complexity.

Exceptions Because changes in color bindings are block scoped, exceptions have minimal impact on thread coloring. Their only impact is the introduction of additional blocks where `Grant` and `Revoke` annotations might appear.

Threads FJ—and so FJC—omits threads. Fortunately, thread coloring analysis need not consider threads directly. All color constraints are expressed in terms of a restriction on the color mapping *of the current thread*. The color environment represents the possible color mapping of all threads that may execute at the point where the color constraint is being checked. Thus, the analysis can treat threads as just another type, whose methods and fields are handled exactly in the usual fashion.

Regions For thread coloring analysis, the Fluid system's notion of a region is a straight-forward extension of FJC's fields. In particular, when processing full Java we actually place color constraints on regions rather than on fields. Likewise, at data accesses we check the constraint on the regions in which the data resides. The checks performed are exactly those shown for fields in FJC. This should not be a surprise—in the full Fluid system, each field has its own region which is the most closely nested region into which the field is mapped. Thus, the only extension needed for full region support is to check constraints from all regions in the hierarchy rather than the constraint on the single region in FJC.

`@Transparent` The as-built Fluid system supports the thread coloring annotation `@Transparent`, indicating that the annotated method or constructor may be invoked from any color environment. This annotation has exactly the semantics of an FJC method or constructor with no explicit color constraint; that is, the constraint is $true$. Note that this annotation establishes an empty color environment inside the annotated method. Such environments are a stringent restriction on the actions allowed in the method body: it may not invoke any method (nor may it reference any data) with a non-empty color constraint. These restrictions, of course, are exactly the same as those on FJC methods with no explicit constraint. In the actual Fluid system, the distinction between an un-annotated method and one that is `@Transparent` is that the annotation represents a deliberate design choice, while the un-annotated method may simply not yet have been examined for thread coloring purposes.

5.5 Future work

In this section we discuss two proposed new features for the thread coloring language. The section closes with a discussion of the steps needed to embark on a proof of soundness.

5.5.1 Color inheritance

The goal of this feature is to support a straight-forward sub-typing relation on color names. This relation must have the obvious effect on color constraints, `Grant`, `Revoke`, etc., so we'll have to modify some of our definitions yet again.

Subcoloring basics

Color extension has entirely unsurprising semantics. Colors are subcolors of themselves, the subcoloring relationship is transitive, and subcoloring is single-inheritance.

$$\mathrm{C} <: \mathrm{C} \tag{5.50}$$

$$\frac{\mathrm{C} <: \mathrm{D} \quad \mathrm{D} <: \mathrm{E}}{\mathrm{C} <: \mathrm{E}} \tag{5.51}$$

$$\frac{\mathrm{ColT}\,(\mathrm{C}) = \mathrm{color\ C\ extends\ D}}{\mathrm{C} <: \mathrm{D}} \tag{5.52}$$

We also need two helper functions: $super$ and sub. Intuitively, these functions return the set of all transitive super-colors and sub-colors (respectively) of their argument, with the exception of the argument itself.

$$\frac{\mathrm{C} <: \mathrm{D} \quad \mathrm{C} \neq \mathrm{D}}{\mathrm{D} \in \mathit{super}\,(\mathrm{C})} \tag{5.53}$$

$$\frac{\mathrm{C} <: \mathrm{D} \quad \mathrm{C} \neq \mathrm{D}}{\mathrm{C} \in \mathit{sub}\,(\mathrm{D})} \tag{5.54}$$

`Grant` and `Revoke`

When granting a color, we must also `Grant` all the colors of which it is a subcolor.

$$\frac{\mathit{super}\,(\mathrm{cname}) = \overline{\mathrm{cn}} \quad \mathit{grant}\,(\chi, \overline{\mathrm{cn}}) = \chi'}{\mathit{grant}\,(\chi, \mathrm{cname}) = ((\exists \mathrm{cname}.\chi') \wedge \mathrm{cname})} \tag{5.55}$$

$$\frac{}{\mathit{grant}\,(\chi, \bullet) = \chi} \tag{5.56}$$

Likewise, when we `Revoke` a color, we must also `Revoke` all its subcolors.

$$\frac{\mathit{sub}\,(\mathrm{cname}) = \overline{\mathrm{cn}} \quad \mathit{revoke}\,(\chi, \overline{\mathrm{cn}}) = \chi'}{\mathit{revoke}\,(\chi, \mathrm{cname}) = ((\exists \mathrm{cname}.\chi') \wedge !\mathrm{cname})} \tag{5.57}$$

$$\frac{}{\mathit{revoke}\,(\chi, \bullet) = \chi} \tag{5.58}$$

Discussion

Without color inheritance, users who declare new colors that are `Compute`-style colors must remember to say that the newly defined colors are incompatible with the `GUI` color. With color inheritance, the user could simply derive her new color from the `Compute` color and inherit the incompatibility "for free." This provides the usual advantages of supporting inheritance for Is-A usage.

The `Grant` and `Revoke` rules specified above are chosen to provide minimal surprise. If a color Is-A `Compute` color it must obviously conflict with all the same things that other `Compute` colors conflict with. We achieve this by simply making sure to `Grant` the parent colors along with the named color, thus adding their conflicts to the color environment. Likewise, when revoking colors, we should clearly `Revoke` all child colors as well. After all, if the thread is no longer a `Foo` colored thread it shouldn't be a `ChildOfFoo` colored thread either.

5.5.2 Parameterized Colors

This section contains a brief and extremely informal discussion of some of the capabilities we should include in a parameterization scheme. This parameterization scheme would support a cleaner approach to handling color constraints for overridden methods such as `Runnable.run()`.

We must support constrained parameterization, for example:

```
/**@Color <X s.t.  AWT => X> */ Runnable runme = ...;
```

Note that the syntax is speculative. The constraint should be read as "X such that `AWT` satisfies X." This annotation would constrain us to assign to `runme` only `Runnables` whose color constraint can be satisfied by `AWT`. Thus, when invoking such a `Runnable`'s `run()` method we would be able to check whether the color environment at the point of invocation satisfies the color constraint on the `run()` method. This improved expressiveness should permit fully-sound handling of overridden methods.

We must support writing a parameterized color constraint for a `Collection`. Suppose we wish to annotate a `Collection` with a color constraint, or better yet a `Collection` that contains data that has a color constraint. We would need to be able to say that the constraint on the `Collection` methods matches the constraint on the data in the `Collection`, and also that the constraint on the `Equals` and `Compare` methods for the data in the `Collection` satisfies the constraint on the data. Finally, we would also need to verify that the color environment at each point of access to the `Collection` satisfies the constraints on the data in the `Collection`. That is, the generic actual color parameter at the point of declaration of the `Collection` should be used as the constraint on the `Collection`'s various methods.

More important still, this latter example needs to work correctly for parameterized types with no constraint at all. That is, an unannotated `Collection` whose contents are also not annotated should consistency check without error.

Formalizing the notion of parameterized colors would require writing both a syntax and a definition of the semantics for parameterized colors.

5.5.3 Proof of soundness

To state soundness in a non-tautological way requires the addition of computation semantics for FJC. Unlike the static semantics, which operate in terms of the color environment at a particular program point, the computation semantics must define the color bindings of the current thread at each step in the computation. Likewise, the FJC computation semantics must also require that color constraints be evaluated with the appropriate color binding. A correct execution would be one in which no color constraint—whether local or global—is violated during execution of the program.

It seems likely that the computation semantics for FJC would be comparable in scope to those of FJ; indeed other than the addition of constraint checking and color-binding tracking they would likely be identical to FJ's execution semantics.

The soundness property to be proven would then be:

> If a program is consistently colored according to our static semantics, execution of that program will not violate any color constraint according to the computation semantics.

This property follows directly from a simpler property:

> If a term is consistently colored and reduces to a second term, that second term is also consistently colored and the reduction does not violate a color constraint.

This latter property is the key to the soundness argument.

To complete our soundness proof, we should also show that in the absence of colors the computation semantics for FJC is equivalent to the computation semantics for FJ. This property—which should be obvious if the FJC computation semantics are correctly specified—establishes that we have produced a sound extension of FJ. The likely similarity of FJC's computational semantics with FJ's computational semantics suggests that the proposed proof of soundness should be comparable in complexity to the soundness proof for FJ.

The addition of parameterized colors to FJC would require an expanded computational semantics and soundness proof. Once again, this appears to be comparable in complexity to the semantics and soundness proof for GJ, as seen in the second half of [31].

Chapter 6

Tool design & usability

6.1 Introduction

In this chapter I discuss the implementation issues—broadly construed—that I encountered while building and experimenting with the module and thread coloring analyses. The discussion begins with an overview of the requirements for the prototype assurance tool, followed by a brief discussion of the architecture and infrastructure of the prototype Fluid assurance tool of which my analyses are part. The section "Engineering the tool" contains a description of the implementation of my analyses that should be sufficient to allow an interested party to duplicate my work. This section also includes informal discussions of the asymptotic performance of my analyses.

The "Engineering trade-offs" section describes the major trade-offs I made during implementation, first discussing my decision to use an incremental lazy-evaluation approach to the analyses and then investigating two points at which I changed the data structures on which the tool operates. The incremental analysis and the first of the two structural changes have been unqualified successes. The second structural change brought improvements in clarity of implementation, but did not provide the expected performance improvements.

The chapter continues with a discussion of some of the usability issues I encountered while experimenting with the prototype implementation of my module and thread coloring analyses. These discussions include descriptions of the solutions, both implemented and not. The penultimate section of this chapter contains a very brief discussion of the issues that must be surmounted to make the prototype assurance tool into a commercial quality implementation. The chapter concludes with a summary of the important lessons learned during implementation of the prototype assurance tool.

6.2 Requirements

The basic requirements for the prototype assurance tool—other than to embody the thread coloring analysis—are quite sparse:

- Use the Fluid project's infrastructure.
- Support case studies including reporting results.
- Address issues of practicability and adoptability.

These basic requirements, however, brought with them a host of other requirements and decisions. Use of the Fluid infrastructure brought many benefits, but also bound quite a few engineering decisions. For example, the Fluid project's tooling supports analysis of Java code only. Fluid is integrated with Eclipse rather than some other IDE. On the benefit side, the Fluid project's infrastructure relieved me of the need to build my own abstract syntax tree (AST), implement my own language front end, do the work of integrating with an IDE, *etc*.

The requirement to support case studies immediately raises the question "Which specific case studies?" The Fluid group's policy is to perform evaluations on other people's code to reduce our own biases [26]. When possible, the evaluated code should be actual fielded production code. This policy drove my selection of examples—I used real applications of substantial size and scope whenever possible. Additionally, as part of the Fluid group I have participated in many case studies in the field; this contact with real programs and real developers has informed my design decisions at every turn.

Addressing practicability and adoptability required that I think more specifically about the needs of an end user. This person would need to analyze real-world programs of significant size and complexity, so the tool should scale well to large programs, handle complex thread usage models, and support both composable analysis and incremental adoptability. In addition, prior research on assurance tools has shown that users are quite resistant to program annotation [18]. While incremental adoption may help ameliorate this tendency, other ease-of-use issues are also important. These include reporting results that make sense, expressing threading models concisely, and, especially, reducing the user effort required to perform thread coloring analysis.

6.3 The Fluid infrastructure

In this section I discuss the general architecture of the Fluid project's prototype assurance tool along with the relevant features of the supporting infrastructure. This discussion is necessary to understand the environment in which my module and thread coloring analyses are implemented.

6.3.1 General architectural description

My thread coloring and module analyses are implemented in the context of the Fluid prototype assurance tool, which is itself a pair of plug-ins for the open-source Eclipse Integrated Development Environment (IDE). Integration with Eclipse allows the Fluid group to concentrate our engineering effort on our analysis and assurance infrastructure; Eclipse provides the Java front-end. Integration with Eclipse is also an important part of our adoptability story. We use Eclipse to provide an incremental and iterative interaction with the assurance tools in the context of an IDE that is familiar to the typical Java programmer.

The Fluid Project's prototype assurance tool is implemented as two Eclipse plug-ins. The *Fluid Plug-in*—of which my analyses are part—is a library of our analyses, data structures, analysis drivers, and other concerns not handled by Eclipse. The plug-in is approximately 180,000 lines of Java code (180 KSLOC). It does not define any Eclipse views, windows, or other user-visible GUI items. These are provided by Fluid's *Assurance Plug-in*, which defines the "Fluid Verification Status" window and provides the means for an Eclipse user to run assurance analyses and to interact with the results. This plug-in is relatively small, only a few thousand lines of code. Figure 6.1 shows the general relationship between the two plug-ins and the Eclipse environment.

The programmer—that is, the Eclipse end user—interacts with the Fluid Plug-in via the "Fluid Verification Status" window provided by the Assurance Plug-in. Each time she saves a file (or some files) that compile successfully, Eclipse notifies the Assurance Plug-in that files have been modified. The plug-in converts the modified files to the Fluid abstract syntax tree (AST), and then invokes analyses appropriately. Each analysis may run either incrementally on the AST for the modified files or globally on the entire AST, reporting errors and assurances that are then displayed in the Fluid Verification Status window.

More specifically, in addition to providing our tool's user interface, the Assurance Plug-in bridges the capabilities of Eclipse and the Fluid Plug-in. The Assurance Plug-in drives our analyses by querying Eclipse about the workspace, obtaining resource handles to all the Java files in the workspace, and subscribing to Eclipse's file update notifications to decide what to re-analyse. For each Java file, the Assurance Plug-in has Eclipse parse it into an Eclipse AST, from which the plug-in creates a parallel AST in our own representation. The Assurance Plug-in then invokes each analysis—the code for which resides in the Fluid Plug-in—on this parallel AST. The analyses report results back to the Assurance Plug-in via callbacks.

It may appear to be redundant to build our own AST when Eclipse already maintains one. We were motivated to use our own AST—which is stored in a structure call the IR (see Section 6.3.2)—for several reasons. From the point of view of expediency, many of our analyses were written prior to our use of Eclipse and are based on our AST representation. More to the point, several of our analyses are control-flow–based and therefore require control-flow graphs (CFGs), which Eclipse does not provide. We already had a framework for generating CFGs from our own ASTs. As a project, we made the engineering decision to write a compo-

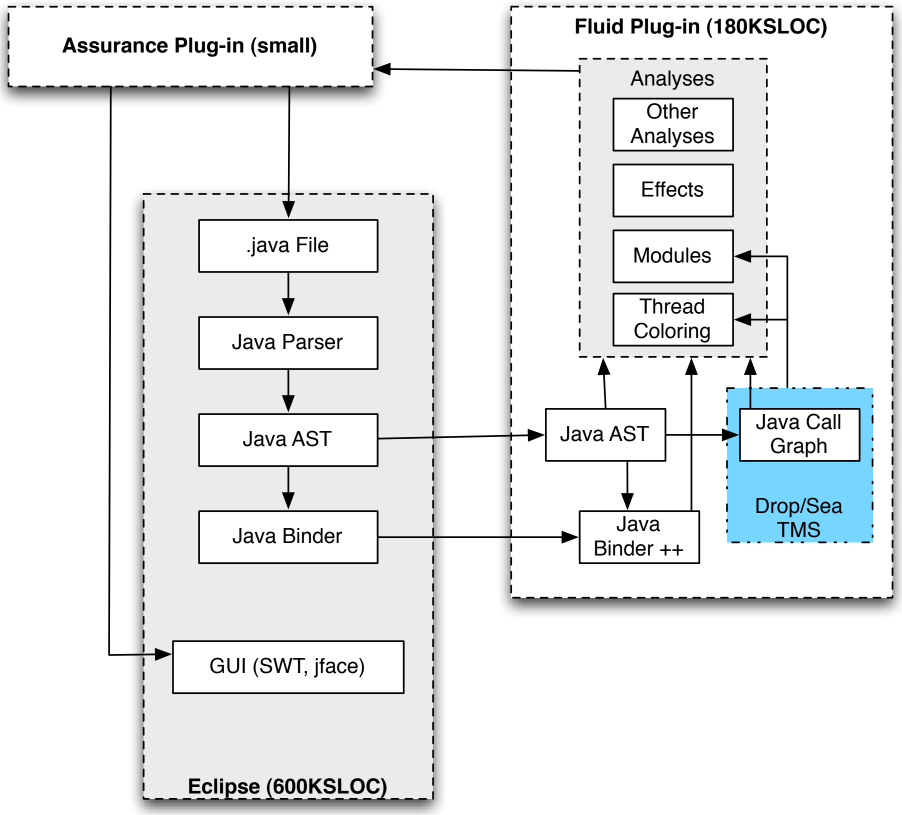


Figure 6.1: Data flow in the prototype Fluid assurance tool

nent that translates Eclipse ASTs into Fluid ASTs so that we would not have to modify already existing and working analysis code.

Our decision to continue to use our own AST representation was additionally motivated by longer-term project goals of developing new techniques for (1) visualizing and browsing source code and models of design intent and (2) managing program evolution. An IR-based AST is necessary for both. As we shall see, versioning is intrinsic to the IR—but is not native to Eclipse—and thus an IR-based AST trivially supports evolution management. Our project has also developed a model–view–controller (MVC) framework on top of the IR that supports elaborate control over how models, *e.g.*, Java programs as ASTs, are displayed. Eclipse provides an elegant and extensive set of GUI components that assist us in the ultimate rendering of our models, but does not provide a suitable MVC framework. Thus supporting within Eclipse our project's visualization research still requires the translation of Eclipse ASTs into Fluid IR representations.

6.3.2 Data representations

The Fluid group's plug-ins use many special purpose program representations in parallel to those provided by the Eclipse API. These data structures are built on top of a data representation framework that we call the Fluid Internal Representation, or IR. This framework predates our work with the Eclipse IDE and provides sophisticated versioning and persistence features that, while unused by our current prototype tool, are critical for the project's longer term goals of studying tools that manage program evolution. I first briefly describe the IR framework, and then describe the program representations built on top of it.[1]

The IR

The IR models general purpose data using a modified version of the standard ternary representation: unique identifiers, attributes, and values. The modifications we make include (1) abstraction to structured entities such as trees and directed graphs, (2) ultra–fine-grained tree-structured versioning, and (3) persistence. There are also several other features of the IR not used in the Fluid group's prototype assurance tool.

Our implementation represents the unique identifiers of the ternary representation as objects that implement a distinguished interface: `IRNode`. This interface defines a small set of operations for getting and comparing identity, and for getting and setting the value associated with that identifier and a specific attribute. Any unique identifier may be given a value for any attribute. Attributes and identifiers can be created dynamically.

Attributes are simply containers for values indexed by unique identifiers; they are represented by objects that are maps from unique identifiers to values. Attributes may also be named and typed. A type in this case refers to an IR data type, not the Java notion of type. IR types are represented by objects implementing a special interface, that includes methods for determining if an object should be considered an instance of the type and for persisting members of the type. The framework provides a set of primitive types including the standard scalar types, *e.g.*, integer, string, etc., as well as several complex types such as sequences and records. Unfortunately, using this kind of meta-level type system means that programming for the IR involves an abundance of type casts and, in general, is unable to take advantage of the type safety provided by the Java programming language. Now that Java supports generic types, the situation is somewhat improved, although a large portion of our code still awaits updating.

Complex structures are built in the IR by considering a given set of identifiers and a given set of attributes to define the scope of the data structure. For example, trees can be represented by mapping each node in the tree to a unique identifier, by defining "parent" and "child" attributes, and by managing the values of those attributes as appropriate to maintain the structure of the tree. Maintaining representation invariants over such a diverse collection of objects is difficult. The IR, therefore, contains classes that provide familiar high-level APIs, implementations of which manage and abstract the underlying attributes. For example, the `Tree` class provides expected tree methods, such as `getParent`, `getChildren`, `setChild`, and `depthFirstSearch`; these methods manage the "parent" and "child" attributes "under the hood." Specifically, the IR provides a sophisticated library of graph classes built on top of the identifier–attribute abstraction.

[1] I am indebted to Dr. Aaron Greenhouse for providing the original version of this discussion of the Fluid IR, and of how the Fluid project uses it to represent Java code. This chapter would be much weaker without his assistance.

Data in the IR actually have a third component: version. Most simply, a complete identifier–attribute—version triple represents a particular value at a particular point in time. Any change to any value produces a new version. The IR maintains a pointer to the current version, which may be queried and reset to a previous version. Our implementation is transparently optimized by not remembering any versions that are never asked for. An additional feature of IR versioning is that the version space is tree-structured rather than "time-ordered" linear. A tree-structured version space allows for experimentation by providing the ability to return to a previous version and make changes to the values, which starts a new branch in the version tree. The IR provides a shadow model of the version space implemented as an IR tree data structure.

Finally the IR supports persistence: the storage and restoration of subsets of in-memory IR structures to and from the file system. The intention is that these subsets be defined along the lines of components and other higher-level abstractions. Persistence preserves identity; if a particular identifier is persisted into two different, but overlapping, subsets of data, then even if the two subsets are loaded into two different Java Virtual Machines (JVMs) the identifier is preserved in both JVMs. Identity is even preserved through structures such as Java `ObjectStreams`. The implemented abstraction is "all identifiers always exist."

It is important to note that the IR is optimized for flexibility in terms of the variety of data being represented and for maximum sharing of data in the presence of fine-grained versions. Ease of programmer use and runtime performance when accessing the data of any specific version were distinctly secondary goals.

Representing Java programs

Fluid's prototype assurance tool represents a Java program using an augmented abstract syntax tree (AST). These trees are stored in the IR. They are represented as IR trees containing an additional "operator" attribute. In addition to describing the role of the node in the program, *e.g.*, "if statement," "field reference expression," or "method declaration," the value of this attribute constrains the number of children the node may have and dictates what their operators may be.

Our control-flow graphs (CFGs) are built from IR-based ASTs following a "cookie cutter" design pattern in which each node in the AST, via its operator value, defines a control-flow component that has one or more "holes" in it filled by the node's children. More specifically, our control-flow graphs contain explicit edges and nodes, and support traversals bidirectionally. Analysis results are stored on the edges. The CFG framework is not Java specific: language specific control-flow components are built out of primitive node types that abstractly determine how lattice values propagate among subcomponents. The framework provides a generic work-list–based control flow algorithm that is parameterized by a transfer function that is both language and analysis specific, and that determines the specifics of how a particular operator affects the incoming lattice value.

Each CFG is itself represented as a data structure in the IR. Its implementation allows sharing of CFG nodes for those portions of an AST that did not change from one version to the next. This enables extremely parsimonious memory use over fine-grained version changes. Once again, runtime performance when using the CFG for any particular version was not a design goal.

The "Drop–Sea" truth maintenance system

The Fluid Plug-in includes a truth maintenance system (TMS) created by Halloran [29] called "Drop–Sea"—Drops of information in a Sea of Truth. We use the TMS to support reporting, as a blackboard for holding partial (or local) results from our analyses, and as an engine for proving global consistency of results. In addition, the TMS provides simple support for incremental analysis.

Each "drop"—implemented by extension of the root drop type `Drop`—contains whatever data the analysis writer needs it to hold, along with a collection of references to dependent and deponent[2] drops. Drops also have the notion of consistency. When a drop is marked as "inconsistent" it is first removed from the deponent set of its dependent drops, then each of its dependent drops is also marked inconsistent. This latter operation is accomplished by calling the drop's `markInconsistent` method. This method's default implementation simply marks the drop as being inconsistent and returns an indication that it did so; this is equivalent to depending on the AND of all deponents' consistencies. Drops are free to override this implementation with

[2]"deponent *n.*^{2}.: One who deposes or makes a deposition under oath; one who gives written testimony to be used as evidence in a court of justice or for other purpose." *The Oxford English Dicionary.* 2nd ed. 1989. Oxford University Press.

a more complicated computation that considers the nature of the dependency and of the data represented by the drop. Thus, it is possible to implement a drop that should remain true while any of its deponents is true; this is an OR-dependency. More complex dependency models are also possible.

The TMS does not store its data in the IR; it uses ordinary Java nodes and collections. As a consequence, the TMS can only be valid for the "latest" version in the IR. We consider the TMS to work with "unversioned" IR; an update to support versioned IR remains as future work for the Fluid group.

The call graph

Both my module and thread coloring analyses depend on the presence of a call graph (CG). Because the existing Fluid infrastructure did not provide a Java CG, I implemented a simple one. The CG contains one node for each method-like entity in the Java AST, including both methods and constructors, whether explicitly or implicitly present. It also includes abstract methods and methods that are declared in Java `Interface`s. The CG is built and rebuilt incrementally. See 6.5.1 on page 154 for details.

The call graph includes nodes for library and framework methods, and for methods whose implementation is either not loaded or not available. It thus represents partial information about the program. CG nodes are represented as drops in the TMS; thus, each CG node is an extension of class `Drop`. These drops contain the callee and caller sets, which are implemented using ordinary Java `Collections`. Callee sets include all known methods that may be invoked from any call site inside the method. The current implementation determines this information based on the declared type of the statically invoked item, including all known overrides or interface implementations of that method. There are well-known optimizations that reduce the size of the set of possible overrides or implementations based on the actual types of the reaching definitions of the objects whose methods are being invoked. However, even though such optimizations would shrink the call graphs and provide more precise analysis, I chose not to implement them. They are not necessary for correct operation of the call graphs and would have increased implementation effort. I also chose not to store transitive may-invoke information locally in each CG node because it is trivially computable from the CG. I chose to store caller sets locally at each CG node; these sets are the union of all CG nodes whose callee set includes this node.

It is important to note that the decision to use *all known* overrides and implementations rather than "all possible" overrides and implementations brings with it a requirement on the end user of my thread coloring analysis. The code presented for analysis as part of any particular module must either

1. Include all possible overrides and implementations, thus ensuring that "all known overrides and implementations" and "all *possible* overrides and implementations" are one and the same.
2. Or include annotations on the known overrides and implementations that correctly provide an analysis cut-point, thus ensuring that the missing information does not cause incorrect results. This requirement is no different from the requirement that methods at module boundaries provide annotations that suffice as a cut-point.

6.4 Engineering the tool

In this section I present enough details of my implementation to allow an interested reader to duplicate its function. This discussion includes a description of the dependency relationships between partial results, but does not otherwise discuss incrementality. Section 6.5.1 on page 153 contains my discussion of the incremental aspects of my analyses. In this section I also highlight some troublesome aspects of the implementation and present informal discussion of the asymptotic performance of each part of the analysis. These performance discussions are *not* based on rigorous analysis. Rather, they are back-of-the-envelope estimates based on prior experience and ordinary engineering rules of thumb. Think of them as being akin to a typical engineering discussion of algorithm selection, not as research into algorithmic performance. That said, I believe that the performance discussions include sufficient detail to permit the construction of a more rigorous analysis by an interested party.

6.4.1 Infrastructure processing

The Assurance Plug-in and associated infrastructure manage the conversion of the Eclipse AST into our Fluid AST (hereafter eAST and fAST when the distinction matters). They also manage early processing of user annotations, during which they perform the pattern matching for scoped promises and place derived promises in the fAST. Additional processing steps include placing non-scoped promises in the fAST, and building TMS "drops" for each compilation unit.

The Fluid Plug-in's infrastructure cooperates with the individual analyses to build the AST that represents user-written annotations. Specifically, the infrastructure parses the annotation and invokes a special part of the analysis code whose duty is to check for obviously broken annotations. In most cases these are simple sanity checks, as full semantic analysis of any given annotation may well require greater context than is available during this early processing. It is important to note that this early handshake between the infrastructure and the analysis code also provides each analysis writer with the opportunity to get his hands on every user-written annotation belonging to his analysis very early in the overall analysis process. It is quite common for analyses to save references to annotations in `Collections` in order to have inexpensive access to all annotations of particular types at a later time, without needing to traverse the entire fAST to find those annotations.[3]

The Assurance Plug-in offers each analysis the opportunity to be invoked at three different times: before processing any compilation units, once for each CU that has changed, and after all CUs are complete. In addition, the Assurance Plug-in tracks dependencies between analyses and guarantees that no analysis will be invoked until all other analyses on which it depends have completed their work.

6.4.2 Module analysis

I now describe the module analysis, which is implemented as a single Fluid analysis phase. It has three primary jobs: to detect any module annotations that violate the module system's rules; to detect any cross-module references that violate module visibility rules; and to prepare the tool's internal data structures for later analysis phases.

As described above, the Assurance Plug-in drives processing for all analyses, and offers three points of action: at the beginning of the analysis, for each new or modified compilation unit, and at the end of the analysis. Module analysis proceeds in two sub-phases:

1. Building a model of the module hierarchy, while simultaneously checking that the expressed module hierarchy follows the structural rules given in Section 3.4.4 on page 45.
2. Checking that each type, field, and method reference in the program is permitted by the visibility rules given in Chapter 3.

The first sub-phase runs at the beginning of the module analysis, before processing the ASTs for new or modified compilation units. It begins by building a model of the module hierarchy, while simultaneously checking to ensure that the expressed module hierarchy follows the structural rules given in Section 3.4.4 on page 45. This is accomplished via iteration over a collection of all annotations that declare modules (*e.g.,* "`@Module`", "`@Module ... for ...`" , "`@Vis`", and "`@Module ... contains ...` "), followed by iterating over all annotations that structure the module hierarchy (*e.g.,* "`@Module ... contains`", the simple dotted names from `@Vis` annotations, and, when they are implemented in the future, the "`@Export ... to ...`", "`@NoExport ... to ...`" and "`@ForbidImport ... from ...`" annotations). While processing each annotation, the analysis modifies the hierarchy to reflect its contribution to the picture. If the contribution introduces a violation of the rules it reports an appropriate error.

When this processing is complete the analysis has built a model of the shape of the hierarchy, and has reported any errors in the module structure. Further, each module in the hierarchy has a symbol table representing its exported interface.[4] Armed with the structural model and associated symbol tables, the analysis is now prepared to check the individual references in the program code.

The second sub-phase is the error checking pass; this sub-phase operates on a per-compilation unit basis, as driven by the Assurance Plug-in. The analysis begins by determining and recording which module contains

[3]Any analysis writer who does so must, of course, arrange to remove obsolete annotations if and when the CU containing them is replaced. My code uses the Drop–Sea TMS to assist with this process; other analysis writers have used alternate techniques. See Section 6.5.1 on page 153 for additional details.

[4]This will include exceptions to the default visibility (see Section 3.4.6 on page 56), when the relevant annotations are implemented.

the CU being processed. Next, it uses the Visitor pattern to traverse the AST for that CU. The action points during the traversal are type, field, and method reference points inside the current CU. At each such reference point, the analysis asks the Fluid Binder to find the specific AST node that is referenced. From this, it finds the referenced item's enclosing CU, and thence the module in which the referenced CU is placed. A simple predicate then suffices to determine whether the referenced item should be visible from the point of reference. The result of this predicate depends on both the module annotations of the referencing CU and the module annotations of the CU containing the referenced entity; the analysis sets the dependencies of the result drop accordingly.

There are two distinct varieties of errors detected by the module analysis. Visibility errors occur when some piece of code refers to a Java entity that it is not permitted to see according to the module visibility rules. These errors need not prevent further processing. However, when the shape of the module hierarchy does not obey the rules from Chapter 3, experience has shown that further processing is highly likely to cause later analyses to fail. Accordingly, in these cases I have chosen to terminate analysis processing.

Asymptotic performance

Building the module hierarchy runs so quickly that it hardly seems worth estimating its asymptotic performance; it took less than 0.5 seconds on a 2GHz Power Mac G5 for the largest example encountered to date by members of the Fluid group.

Rule-of-thumb estimation of the asymptotic performance of the visibility error checking pass is slightly more complicated. First, there is a pass across the entire program, with action at each reference to a Java entity; visiting the entire program clearly must be at least linear in program size. The actions taken at each entity include:

- Binding the name of the entity. I have not estimated the cost of this action, as its implementation is out of my control. The cost must be acceptable, however, as it happens repeatedly both throughout Fluid and also throughout Eclipse.
- Looking up the module containing the referenced entity. $\mathrm{O}\left(\lg n\right)$ in the number of modules.
- Symbol-table lookup in the module containing the referenced entity. $\mathrm{O}\left(\lg n\right)$ in the number of entities forming the visible interface of that module. This number grows proportionally to program size, at worst. Carefully implemented hashing schemes may reduce this to linear cost in practice.

Combining these, my back-of-the-envelope estimate is that the entire module analysis takes time no worse than $\mathrm{O}\left(n \lg n\right)$ in program size. It is indicative that module analysis for Electric's 140KSLOC completes in fewer than ten seconds on the aforementioned Power Mac.

Building the static call graph

All the analyses that use the static call graph depend on the module analysis. Furthermore, the module analysis must already visit every call site in every CU that is new or changed. Accordingly, the module analysis builds the CG while performing the error checking traversal. There is no inherent requirement that this be so; it is merely a convenient place to do the work.

Building the call graph requires two additional interfaces, one from the Fluid Binder and one from the call graph support code. At each call site, the CG-building code asks the binder for a `Collection` representing all known overrides and implementations of the method or constructor being invoked. It then builds or updates the call graph by iterating over this `Collection` noting the existence of a call from the current method to each of the methods returned by the binder. The call graph support code responds to this information by adding the caller and callee nodes to each other's caller and callee sets in the obvious fashion.

6.4.3 Thread coloring analysis

I now describe the thread coloring analysis. Its jobs are to perform thread color inference, to report inconsistencies between the expressed thread usage policy and the as-written code, and to report the computed color environment for each method in the code being analyzed. Thread coloring analysis runs in three complete passes, each of which is implemented as a standard Fluid analysis. The first two of the three passes perform set-up work for the analysis; the third pass performs the inference, and the error checking and reporting.

General design and implementation choices

I chose to use ordinary Java libraries and types to implement the bulk of my supporting data structures, rather than storing those supporting structures in the IR. As a direct result of this decision I lost the opportunity to experiment with fine-grained versioning, but also avoided the need to understand the versioning aspects of the IR. As additional benefits, I gained more convenient predefined interfaces and better run-time performance.

Thread coloring analysis represents all Boolean expressions as Binary Decision Diagrams (BDDs). From an algorithmic point of view, this allows it to support general Boolean expressions both for color constraints and for the current color environment, thus allowing reasoning about sets of color bindings rather than about a single color binding as in the initial implementation. Using BDDs offers many additional benefits. Typical BDD operations take time that is linear in the number of variables in the expression. By their very nature BDDs are efficient in terms of memory usage. There are many pre-existing and actively maintained BDD packages to choose from. I chose to use the open–source SableJBDD library, because

- It is written in Java and so integrates easily with the Fluid Plug-in.
- It is open source, so I need not depend on the author for all bug fixes.
- It supports garbage collection of unused BDDs and BDD variables. Unlike other Java BDD libraries, there is no need for reference counting or explicit deallocation. This is a significant advantage in terms of development effort.
- It is licensed under the Lesser Gnu Public License (LGPL). This license permits no-cost integration into both research and commercial tools. This avoids encumbering my implementation with inconvenient or overly restrictive license terms.

The current implementation of SableJBDD is not optimized for speed, and so is notably slower than other BDD packages for Java. This does not matter for my needs: when considered by the standards of most BDD users, my expressions are small and infrequent. Thread coloring analysis of even a large program presents little or no challenge for a BDD library.

Each thread coloring analysis pass uses Halloran's Drop–Sea TMS [29] both for consistency management and as a blackboard to hold analysis information and partial results. A key decision I made is to represent most of the data computed by the thread coloring analysis as "drops," and to perform incremental recomputation when possible:

- Every user-written annotation is represented by data stored in a drop. This drop serves as a deponent for everything we compute based on that nugget of information.
  - Each kind of user-written annotation has its own drop sub-class; the annotations are stored as instances of these sub-classes.
  - Annotations that contain Boolean expressions represent those expressions as BDDs stored in the drop.
- Every method-like Java entity gets a "computed color constraint" and a "calling color environment" drop. These drops obtain their initial values from the user annotations; their values are then refined during thread coloring analysis.
- Data regions with color annotations get a "user constraint" drop and a "accessing color environment" drop. The user constraint is initialized with the color constraint specified by the user, if any. The accessing color environment drop is computed by thread coloring analysis.
- I use `ResultDrops` for reporting. This kind of drop is the standard mechanism whereby Fluid analyses report their results.[5] `ResultDrops` have a Boolean flag that indicates local consistency. They can also depend on other result drops and let the TMS compute global consistency.[6] Thread coloring analysis uses local consistency only, because it does its own global consistency proof.

In the discussion that follows, all drops involved in thread coloring analysis are referred to as "color drops" when the particular subclass is not important, or by their type name when the particular subclass matters to the discussion. I describe here the operation of thread coloring analysis without addressing incrementality; the additional challenges raised by incrementality are addressed in Section 6.5.1 on page 153.

[5]The main engineering of result drops and reporting was done by Halloran, with assistance from the research group as a whole.
[6]See [29] for details.

First pass: Imports and name binding

Thread coloring analysis is split across three analysis passes, each of which must complete before the next may proceed. The first of these passes processes `@ColorImport` annotations and performs name binding for all color names found in color annotations. This first analysis pass is purely local in character: the tool visits each compilation unit (CU) at most once.

As a convenience for the user, thread coloring analysis supports both fully qualified references to colors and the use of simple names. Simple names refer either to a local color definition, or to a definition that has been imported from elsewhere by way of a `@ColorImport`; thread coloring analysis must bind these names to the appropriate definition before proceeding. To bind the names, it must process the color imports.

The first pass builds a `ColorImportDrop` in the TMS for each `@ColorImport` annotation it processes. This drop serves to encapsulate the fact that drops built in the current CU depend on the definitions imported from elsewhere by the annotation, and must be invalidated if those definitions are modified. Thread coloring analysis represents the dependency by adding the remote definitions to the `ColorImportDrop`'s deponent set. Later color drops built in the current CU will depend on the `ColorImportDrop`, and will be invalidated when appropriate.

The first pass saves time and space by delaying processing of compilation units whose source is not loaded for analysis until it sees a reference to some Java entity in the compilation unit. This lazy processing permits the first pass to avoid processing most of the Java libraries; it spends time and storage only on those library units that are actually referenced by loaded code. The first pass applies the same logic to compilation units whose home module was loaded "As-Spec" rather than "As-Source" thus avoiding additional needless processing.[7]

This lazy evaluation scheme provides a significant savings in time and storage. Although the runtime of the first pass is essentially linear in program size, the Java libraries and runtimes are quite large; furthermore, a typical program imports many times more Java entities than it defines locally. Even though the Fluid infrastructure loads only the specification information for the Java libraries, the time to process those specifications is noticeable. Similar logic applies to avoiding the processing of unreferenced specifications from modules loaded "As-Spec." It is only a linear speed-up, but it still matters; when analyzing 50KSLOC or so the omitted work would make up around 50% of first-pass analysis time.

Asymptotic performance The main processing of the first pass is fundamentally linear in nature; it visits each CU and AST node at most once. However, as part of my implementation, I maintain various sets, hash tables, and other book-keeping data. No operation on this book-keeping data need be worse than $\mathrm{O}(\lg n)$ in program size, nor need the number of such operations grow any faster than the program size, so the entire pass should be no worse than $\mathrm{O}(n \lg n)$ in program size. Further, I suspect that a carefully written implementation could remove the $\mathrm{O}(\lg n)$ factor, perhaps through the use of carefully sized hash tables, and so achieve $\mathrm{O}(n)$ in the typical case.

Second Pass: Rename expansion, constraint inheritance, and drop building

The second pass does all the remaining work required to get ready for the color inference algorithm and error checking. Specifically, it performs expansion of `@ColorRename` annotations, builds drops for user annotations, and does inheritance processing. Second pass processing is split into two traversals: one processes all of the non-method-like entities in the CU (*e.g.,* types, fields, *etc.*), the other is invoked separately for each method-like entity.

Rename expansion When writing color constraint annotations for case studies, I quickly noticed that I was writing similar expressions over and over. Eventually this became so tedious that I designed and built a simple renaming facility to make the case studies go faster. By writing "`@ColorRename foo` **`for`** `((tom & !dick)| harry)`", for example, a programmer declares that the name `foo` should be expanded into the given expression "`((tom & !dick)| harry)`". Of course, the expanded expression may itself use names defined via `@ColorRename` annotations; these must be expanded in turn. Color renamings are found either locally—within the current CU—or are imported from elsewhere via a `@ColorImport`

[7]See Section 3.5.1 on page 61 for more information on this distinction.

annotation. The first pass has already made imported renames visible within the correct compilation units; the second pass must now expand them. Only one portion of this expansion process is non-obvious: drops that contain expanded names must depend on either the drop for the defining color rename annotation if the declaration is local, or on the drop for the appropriate color import if the declaration is not local.

Drop building After expanding renames, the second pass builds the actual drops in the TMS that represent user and library color constraint annotations. This process involves filling in the drop with the text of the annotation and creating the BDD that represents the Boolean expression from the annotation. As mentioned in the previous section, these drops must depend on the appropriate `ColorImportDrop`s as needed to reflect their dependency on imported definitions.

Inheritance processing The second pass's final task is to perform inheritance processing for color constraints. As described in earlier chapters, the default behavior is for each method to inherit its color constraint from the method it overrides, if any, or from the interface method it implements, if any. This inheritance is transitive. Local user-written color constraints, however, override inherited color constraints. To effect this inheritance, at the declaration of each method-like entity the tool looks up the chain of overrides and implementations until it either finds a color constraint, reaches the top of the chain, or finds a method in the chain that has already been processed. The color constraint found, if any, is propagated down the chain to the current method. The tool stores the color constraint as a drop associated with each method. Each such drop depends on the color constraint drop for its immediate parent in the chain. Whether or not a color constraint is inherited, the final action for each method is to initialize its computed color constraint to be a copy of the color constraint we have just computed, or to `true` if there is no color constraint. The color drop that holds the computed color constraint is set to depend on the source from which we initialized it.

Careful readers will have noted that this algorithm poses a problem with respect to processing order. Methods overridden or implemented by the current method are likely to come from some remote CU, that is a CU that does not contain the current method. Nothing in the description of the second pass presented so far guarantees that such remote CUs have already been through the second pass processing; thus the remote CU may not yet have had its renames expanded and its drops built. Furthermore, looking upwards along the inheritance and implementation chains may result in visiting multiple remote CUs, each of which may or may not have been through second-pass processing prior to our examination of it. The second pass solves this problem by guarding its processing with a pair of flags in the AST; one for the compilation unit and one for each method-like entity. These flags are clear when initialized, and are set for each CU or method when its processing begins. During inheritance processing, the second pass checks the flag for each method it visits and for the CU containing said method; if the flag is clear, the second pass recursively invokes its processing on that CU or method, setting the flag as it goes. This ensures that second pass processing for the remote CU or method is complete before the second pass tries to use the results of the processing; using the flag cuts off infinite recursion and enables the second pass to ensure that each entity is processed only once no matter how many times it arrives there. Note that as with the first pass, this mechanism also supports lazy evaluation of library and "as-spec" entities.

Asymptotic performance As with the first pass, the bulk of second pass processing is fundamentally linear in nature, with an additional $\mathrm{O}(\lg n)$ factor in program size to allow for the overhead of book-keeping structures. This would appear to yield performance that is no worse than $\mathrm{O}(n \lg n)$ in program size.

Sadly, that's not the whole story. Building BDDs is potentially exponential in the number of variables, because each BDD effectively contains both the disjunctive normal form (DNF) and the conjunctive normal form (CNF) representation of its expression; conversion between these forms is well known to be exponential in the worst case. Fortunately, the typical case performance for building BDDs is linear in the number of BDD variables. Even more fortunately, the BDDs used for thread coloring analysis tend to be quite small, both in number of terms and in number of variables. The largest number of variables I have observed to date is less than thirty. This is a miniscule number in the world of BDDs.

The worst-case performance of the second pass is exponential, but the typical case is no worse than $\mathrm{O}(n \lg n)$ in program size. I have not seen any exponential behavior to date. Further, I suspect that a

```
while (worklist_not_empty) {
  m = worklist.getNext();
  if (!m.hasDeclaredConstraint()) {
    m.constraint = m.callingEnvironment;
  }
  currentColorEnvironment = m.constraint;
  // process m with the current color environment,
  // possibly adding invoked methods to the worklist
  m.process(currentColorEnvironment);
}
```

Figure 6.2: Work-list processing pseudo-code

carefully written implementation could remove the $O(\lg n)$ factor, perhaps through the use of carefully sized hash tables, and so achieve $O(n)$ in the typical case.

Third Pass: Constraint inference, error checking and reporting

The third pass first performs the color inference algorithm, then does a single linear traversal of the entire program checking for errors and producing assurance results.

Color inference The color inference algorithm computes:

- The color environment from which each method is called, known as the "calling color environment"
- The color environment from which each colored data region is accessed, known as the "accessing color environment".

For methods that do not have a declared color constraint, the calling color environment is also the color constraint. For colored data regions that do not have a declared color constraint, the accessing color environment informs us of the conditions under which the data region is accessed. Color inference differs from typical flow-based algorithms in that it is concerned with flow on the call graph (CG) rather than with flow within the CFG for a single method. It is implemented as an iterative work-list based algorithm, with a single per-method processing component. The "join" operation is the OR of Boolean expressions.

The third pass initializes the work-list with all methods that have both a declared or inherited color constraint and for which the Fluid infrastructure has loaded the AST representing the body of the method. Processing of the work list proceeds as shown in Figure 6.2.

Processing of each method consists of walking the AST for the method simulating changes, if any, to the current color environment and propagating that environment to all potentially invoked methods. Recall from Section 5.4 on page 135 that only `@Grant` and `@Revoke` annotations can change the current color environment; these annotations can occur only at block boundaries. Thus, any method that contains neither `@Grant` nor `@Revoke` annotations executes entirely in the color environment present on method invocation. This property enables our first optimization: the third pass need not actually traverse the bodies of such methods. Rather, it simply looks up the method's callees in the call graph, and propagates the color environment to those callees directly by OR-ing the current color environment into the computed calling environment for each potentially invoked method.

For methods that contain one or more `@Grant` or `@Revoke` annotations, the third pass must traverse the body of the method. While traversing, at each call site it encounters, it ORs the current color environment into the calling color environment of each of the methods potentially invoked from that call site. When it encounters a block that has `@Grant` or `@Revoke` annotations it pushes the current color environment onto a stack for later restoration, modifies a copy of the environment to reflect the effect of `@Grants` or `@Revokes`, and continues the traversal. On exit from the block, the third pass restores the color environment previously saved on the stack.

There are two tricky parts of this processing. The first is knowing when to add a method to the work-list. Each time the third pass modifies the calling environment of a method it must decide whether the method should be added to the work-list. When the calling environment after the OR is the same as it was before the OR, there is no point in adding the method to the work-list, as no information has changed. When the calling environment after the OR has changed, the invoked method is a candidate to be added to the work-list. A candidate method that has an explicit color constraint, whether user-written or inherited, need not be added to the work-list: such a method *always* has the color environment defined by the explicit color constraint. Methods that lack an explict constraint, however, are added to the work-list because the newly updated calling color environment means that such methods must be re-processed to consider the newly available information.

The second tricky issue is maintaining correct dependencies between drops. When considered as a purely local question, however, maintaining the dependencies is almost trivial. The "trick" in one sentence is "when updating a drop, add that drop to the dependent set of the drops that produced the information." To implement this trick, the third pass maintains an additional datum in parallel with the current color environment: a deponent set for that environment. When starting the traversal of a method, this deponent set is initialized with the drop that represents the constraint for the method whether computed or explicit. When the current color environment is updated due to a `@Grant` or `@Revoke` annotation the deponent set is similarly updated by adding the drops for the `@Grant` or `@Revoke` annotations. Note that a newly encountered `@Grant` or `@Revoke` annotation may replace a prior member of the deponent set because it subsumes all color information provided by that prior member.[8] For example, a newly encountered `@Grant bar` annotation would replace a prior deponent that was either `@Grant bar`, `@Revoke bar` or `@Color bar`; it would not replace a prior deponent such as `@Grant foo` or "`@Color foo | bar`". When propagating the current color environment to an invoked method, the third pass updates the drop for the computed calling color environment of the invoked method to include the deponents of the calling method's current color environment. The final piece of the puzzle is to ensure that the drop for the color constraint on each method depends on the correct deponents. For methods that have a user-written annotation, the deponent is the drop for that annotation. For methods that have inherited a color constraint, the deponent is the drop for the color constraint of the method's parent in the override and implementation hierarchy. For methods that have a computed color constraint, however, the deponent is the drop for the computed calling color environment.

Asymptotic performance of color inference I now discuss the number of iterations required to complete the color inference computation and thus the runtime cost of the inference algorithm. The color inference algorithm appears to be closely analagous to the type inference performed for languages such as ML. The best known algorithms for type inference are exponential in program size in the worst case, but are linear in practice.

I further observe that the inference algorithm looks a lot like typical CFG-based iterative algorithms that traditionally operate inside methods. These algorithms have asymptotic performance that is $\mathrm{O}\left(n \times uc \times (\mathit{loopDepth} + 1)\right)$, where n is method size, uc is the update cost at each action point, and $\mathit{loopDepth}$ is the maximum static loop nesting depth in the method[9]. Because my inference algorithm is operating on the CG rather than the CFG, the loops of concern are those formed by recursion in the CG, rather than those formed by iteration in the CFG. As with loop nesting depth inside methods, the maximum static recursion-depth[10] appears to be limited by a small constant in practice. The action points in the inference algorithm are call sites; the number of call sites appears to grow linearly with program size. The update cost at each action point is a single BDD operation (Boolean-OR). As with other BDD operations, this is potentially exponential in the worst case, but is typically linear in the number of BDD variables actually present in the expression. The number of BDD variables needed for thread coloring analysis appears to be limited by a small constant—I have never seen more than thirty, and typically see far fewer.

I have experimented by analyzing programs that include carefully crafted recursive methods that invoke other recursive methods (and so on) in an attempt to observe non-linear behavior in the analysis. As expected, methods inside such nested recursive loops in the call graph are visited $\mathit{loopDepth}+1$ times. This observation cannot rule out worst-case behavior that is less desirable than I estimate here; it does, however, bolster my confidence in my typical case analysis.

As with the other analysis passes, color inference uses a variety of book-keeping and data access methods whose invocation count scales linearly with program size. Most of these methods should have constant

[8]This replacement is not required for correct behavior. It simply reduces the length of dependency chains, thus possbily reducing the cost of recomputation after source code is modified.

[9]This result assumes structured code, which assumption may not hold for call graphs.

[10]The nesting depth of CG loops due to recursive methods, *not* the dynamic recursive invocation count of such a method.

cost; none of them need be worse than $\mathrm{O}(\lg n)$ in program size.[11] Thus, I estimate that the typical cost of color inference is exponential in the worst case, and $\mathrm{O}(n \lg n)$ in the typical case. Further, I suspect that a carefully written implementation could remove the $\mathrm{O}(\lg n)$ factor, perhaps through the use of carefully sized hash tables, and so achieve $\mathrm{O}(n)$ in the typical case.

Error checking and result reporting When color inference is complete, all that remains is to report results. At this point in processing, every method has a pair of color drops of interest: one holds the computed calling color environment for the method, the other holds the method's color constraint (whether inferred, inherited, or user-written). Similarly, each data region for which color analysis is performed has a pair of color drops representing the accessing color environment and the region's color constraint, if any. At the highest level, checking for color consistency simply requires testing whether or not each color environment satisfies its matching color constraint. The third pass does this by evaluating the Boolean expression $(\mathit{color\ environment}) \Rightarrow (\mathit{color\ constraint})$; a result of true indicates satisfaction, and false indicates an inconsistency.

Sadly, reporting is not quite this simple when there are inconsistencies. The problem is that end users are not happy to be told "there is a color inconsistency somewhere in your program involving a call to this specific method. Good luck finding it!"[12] Reporting the *specific location* of each inconsistency is much more useful, and is certainly necessary for any hope of adoptability. This requires a traversal of the entire program.

The error reporting traversal visits each method in turn, checks to see whether the calling color environment satisfies the method's color constraint (reporting an error if it detects an inconsistency), then processes the body of the method. The error-checking walk across each method body is fundamentally the same as the per-method traversal used during the color inference algorithm; indeed, in the current version of thread coloring analysis it is implemented by the same method. The differences are these: (a) in place of updating the calling color environment of potentially invoked methods, the third pass instead checks to see whether the current color environment satisfies the color constraint of the potentially invoked method; (b) the third pass traverses the body of each method—the optimization for methods that leave the color environment unchanged does not apply here; and (c) the third pass checks whether the current color environment satisfies the color constraints on referenced data regions as well as on potentially invoked methods. During color inference there was no need to consider data references. Note that information on referenced data regions is provided by the Greenhouse–Boyland effects analysis [24]; thread coloring analysis simply applies its findings to the results of the effects analysis.

There is still a little more work to do after the error reporting pass over the program's code is complete: the third pass visits each data region for which it has computed color information. For those regions that have color constraints, it reports whether or not the constraint was satisfied at all references to the region. For those regions that lack color constraints it reports the accessing color environment as information to the user.

Asymptotic performance of reporting As with the previous passes, because the third pass is using BDDs, its worst-case performance is exponential in the number of BDD variables. The typical cost for the BDD operations, however, is linear in the number of variables. As before, the number of BDD variables appears to be limited by a small constant, so the typical cost of each of these operations is roughly constant.

Reporting results requires a linear pass across the entire program; simply visiting each node has a cost that is at least linear in program size. At each method call site and each data region reference the third pass performs a constant-cost BDD operation, and produces a result. Applying my usual back-of-the-envelope $\mathrm{O}(\lg n)$ factor to allow for data structure access and other book-keeping, I arrive at a typical cost of $\mathrm{O}(n \lg n)$ in program size. As before, I suspect that a carefully written implementation could remove the $\mathrm{O}(\lg n)$ factor, perhaps through the use of carefully sized hash tables, and so achieve $\mathrm{O}(n)$ in the typical case.

Volume of results A particular challenge for thread coloring is the volume of results produced. As currently implemented, for the portion of the program under analysis, the third pass produces

- One report item for each method/constructor call site.

[11] Indeed, to the best of my knowledge, none of them need be worse than linear, but I'm trying to be conservative here.

[12] In fact, I wasn't willing to live with this style of reporting when experimenting with the analysis. And any ordinary end user has even less incentive to put up with sub-optimal reporting than I do in my role as analysis developer.

- One report item for each data reference that might be to a color-constrained region.
- One report item for each color-constrained region, reporting over-all success or failure of its constraint.
- One report item for each method/constructor, reporting what its color constraint was and whether all invocations respected that constraint.
- One report item for each colorized region, reporting its computed accessing color environment.
- One report item for each declared color name, reporting whether it participates in any coloring inconsistencies.

In a program that is fully annotated, with color constraints for every data region and every method, this could be multiple report items for each line of code. For example, when processing a large program such as all of Electric, the third pass can produce more than one hundred thousand report items. Both the Assurance Plug-in and the underlying Eclipse infrastructure perform poorly when presented with this number of items. I return to this issue in Section 6.6.1 on page 160, as part of my discussion of usability issues.

6.5 Engineering trade-offs

In this section, I discuss the major engineering trade-offs I made while implementing my module and thread color analyses. Section 6.5.1 describes general issues of incrementality, then continues by describing how my implementation and algorithms differ from the non-incremental description given above. In Section 6.5.2 on page 156, I discuss a pair of changes, each of which involved a major change in the data structures traversed by my thread coloring analysis. For each, I discuss my motivation for change, the surrounding issues, and the results both in terms of performance and in terms of tool attributes such as maintainability and usability.

6.5.1 Incrementality

My decision to implement the module and the thread color analyses using an incremental, lazy-evaluation approach taught me several things about incrementality. By choosing this approach I learned that incrementality need not be difficult, provided that one

- has a framework that supports and enables incremental invalidation of fine-grained partial results.
- reduces each decision about establishing desired dependencies to a simple and most importantly, *local* decision.
- reduces each decision about recomputation strategy to a simple question that can be addressed while invalidating an individual partial result.

The availability of the Drop–Sea TMS [29] inspired me to attempt incremental analysis anywhere I encountered an easy opportunity to do so. Without the TMS, I would have viewed incremental analysis as a distraction from my overriding goal of building the thread coloring analysis. I have not encountered any significant problems with the implementation of the incremental portions of my analyses; rather the difficult bugs have all been either due to fundamental algorithmic issues regarding the thread coloring analysis or due to phase-ordering issues that were entirely independent from the issues raised by incrementality. I believe that the incremental aspects of this implementation have been an unqualified success.

General analysis issues

Incremental analyses must be concerned with three general issues:

1. Invalidating the correct subset of their data when a portion of the program is replaced. This requires establishing dependencies within the TMS that correctly reflect a specific datum's dependence on other data used in the analysis. I have found that identifying these dependencies is easy for local analyses, and has been tractable for the global portions of thread coloring analysis.
2. Processing new program files as they appear, to produce the correct local data.
3. Combining new local data with surviving data to produce the correct result. Depending on the specific analysis, this may require traversal of CUs that have not changed since the end of the previous analysis.

This need arises when transitive invalidation of drops affects data associated with a CU that was itself unchanged, but whose analysis-related drops depended—perhaps transitively—on drops associated with a CU that did change.

I now discuss these three issues in general, followed by a specific discussion of how they affect each of the key analyses I have implemented.

The Drop–Sea TMS provides significant assistance with the first issue. Specifically, it provides a framework for managing dependencies and for invalidating them appropriately. Perhaps most important of all, as described in 6.3.2 on page 143, the Drop–Sea TMS localizes the invalidation logic for each variety of drop; this greatly clarifies an implementor's view of the necessary actions.

The Assurance Plug-in provides a convenient driver that addresses both processing new program files and also detecting the deletion or replacement of existing files. The Fluid Plug-in also tracks these events through the Sea by producing a `CUDrop` for each compilation unit; the Drop–Sea framework invalidates these drops automatically when the abstract syntax tree (AST) for that CU is deleted or replaced. By convention, all drops pertaining to parts of a CU depend on the `CUDrop` for that CU; they are thus invalidated when the underlying CU changes. The Assurance Plug-in also ensures that new CUs are processed when available, thus ensuring that each analysis has the chance to produce local data for those new CUs, and so addressing the second issue. This pattern of dependence on the `CUDrop` for the containing CU is assumed in the discussion of incrementality that follows; it is not mentioned specifically in the sections below.

Combining new local data with surviving analysis data is trivial for analyses that are either purely local, or are entirely confined to a single CU. After all, if the analysis is purely local, its results do not depend on the presence or absence of results from some other CU and so are not affected by the invalidation of that other CU. For non-local analyses this combination step either requires great care in correctly identifying what data must be recomputed or requires that the analysis abandon incrementality and simply invalidate and recompute all of its results.

Building and rebuilding the call graph

Construction and reconstruction of the call graph is fully incremental. The CG models dependencies for purposes of invalidation by following the Fluid convention that each location-dependent drop depends on the `CUDrop` for the CU that contains it. CG nodes are thus invalidated when their CU is replaced.

Because each CG node contains sets of callees and callers, and because those sets may refer to methods in other CUs, the CG support code must take additional action to ensure that invalidation of a CG node leaves behind a valid CG. Specifically, when a CG node is invalidated, it removes itself from the caller set of each of its callees and from the callee set of each of its callers. The resulting graph is correct in the absence of the removed CU. This invalidation action satisfies the requirements for incremental invalidation.

The Assurance Plug-in drives processing of newly created files and of new versions of previously existing files. Local processing for these files produces the local information required by the second incremental rule.

Suppose, however, that the CU was edited rather than deleted. The analysis will see this as the removal of the CU followed by the arrival of a new CU containing some or all of the same methods. It rebuilds the edited CU's CG nodes along with their callee information during the course of ordinary processing (along with the caller sets of the methods they invoke, of course). Without additional effort, however, it will miss updating the CG nodes for methods from unchanged CUs that invoke those found within the edited CU. Specifically, the callee sets of those unchanged methods will not include methods from the edited CU even though references to methods in the edited CU should be present. To solve this problem, the CG support code maintains a `Set` of unchanged methods to re-process. This `Set` is populated with the methods whose callee sets we touched during invalidation of CG nodes, but with those methods that were themselves invalidated removed from the `Set`. Reprocessing the methods on this `Set` fills in the missing pieces of their callee sets, along with the caller information that would otherwise be missing from the edited CU's methods.

Incremental module analysis

Recall from Section 6.4.2 on page 145 that the module analysis has both a local and a global portion. I chose to repeat the global portion of the analysis—that is, the traversal of annotations that form the module hierarchy and building the symbol-tables for that hierarchy—in full each time any code changes. I made this decision partly because the global portion of the module analysis runs quickly enough that there is no significant benefit to be gained by incrementality, and partly to avoid the effort of figuring out how to manage partial deletion and reconstruction of the module hierarchy. I chose, however, to make the error-checking pass over the program's source code fully incremental.

Maintaining the correct dependencies for invalidation both of module annotations and of the results of module analysis turned out to be easy. First consider the conditions that may cause changes in the module annotations for a CU.

- The annotations may change when the CU is edited. Fortunately, all of the annotations and results for a given CU are automatically invalidated along with the CU when it is edited or deleted. Further, the Assurance Plug-in already ensures that the module analysis will be re-invoked for the new version of every edited CU. Thus the analysis has no need to worry about either arranging for the invalidation of annotations in edited CUs or arranging for reprocessing of these CUs and their contents.
- The annotations for a particular CU may change when they are derived from a scoped annotation located elsewhere, and that scoped annotation is edited. The Fluid infrastructure treats this case exactly as though annotations local to the effected CU had been edited, so this case is essentially similar to the local edit case.

The second issue related to dependencies is ensuring that computed results are invalidated correctly. As seen above, this is not a problem for edited CUs. However, a change in the visibility of a Java entity may potentially affect the correctness of all references to that entity from some other CU. The tool must ensure that analysis results for any such reference are invalidated. As mentioned in Section 6.4.2 on page 145, each computed module analysis result depends on both the module information of the referencing CU and that of the CU containing the referenced entity. Because this dependency is recorded in the `ResultDrop`, results in referencing CUs are invalidated when the referenced CU's module information changes. As with the call graph, the module analysis maintains a `Set` of CUs to reprocess. Each time a module analysis `ResultDrop` is invalidated, the CU that contains it is added to the `Set` (assuming that the CU itself is valid). During module analysis, each CU processed as a result of the Fluid framework's ordinary processing is removed from the `Set` of CUs to reprocess. When normal processing is complete, any CUs remaining in the `Set` are processed as though they had been changed.

Incremental thread coloring analysis

My approach to making the first two thread coloring analysis phases incremental is exactly analagous to the approach used for call graph building and for the module analysis, differing only in the details of the specific dependencies between the drops. The third pass's iterative color inference algorithm poses an additional challenge, however. My current approach is to allow invalidation of computed calling color environment results based on the invalidation of the user-written annotations from which they originate. This invalidation propagates transitively through the methods in the call graph via the dependent/deponent links in the drops that hold this information.

For the iterative color inference algorithm itself, the only difference between initial computation and re-computation is that for re-computation passes I add every method with surviving calling color environment information to the work-list—the *surviving methods*—in addition to the usual "all methods with specified color constraints." During work-list processing, the tool computes calling color environment information as described in Section 6.4.3 on page 150; this information is propagated to possibly invoked methods using a Boolean-OR operation. Any surviving method will never again be added to the work-list because the newly computed calling color environment information will not change the surviving calling color environment information. The presence in the work-list of the surviving methods guarantees that their current color environment will be correctly propagated to any non-surviving methods they may invoke.

This approach is sufficient for correct recomputation of the calling color environment, and thus for inference of color constraints. It is more efficient than recomputing from scratch, in that surviving methods that participate in loops in the CG are processed only once rather than static-loop-depth-plus-one times. That said, this approach incurs more computation than is strictly necessary. Specifically, each surviving method is guaranteed to be removed from the work-list and processed exactly once, even though many surviving methods may not require re-processing at all. I believe that the correct initial population of the work-list is as follows:

- All non-surviving methods that have explicitly written color constraints, plus
- All surviving methods that have at least one callee that is non-surviving.

Changing to this strategy would avoid re-processing of portions of the call graph where no change has occurred, while still ensuring that all non-surviving methods wind up with correct calling color environment information.

Color analysis results reporting is not incremental at all. The current implementation of the third thread coloring analysis pass invalidates all color analysis results and rebuilds them from scratch, incurring the full cost of the global traversal described in Section 6.4.3 on page 150. I believe that this error reporting phase could easily be implemented as an incremental algorithm, but I have not yet done so.

Enduring lessons

Linear speed-up due to avoided work can be worth pursuing, as long as the effort required to achieve the speed-up is reasonable.

Incrementality need not be difficult, given rich support for managing dependencies and a suitable approach to establishing, maintaining, and rebuilding consistent analysis information. The Drop–Sea TMS, in combination with the Fluid infrastructure as a whole, provides the needed support for managing fine-grained dependencies and for storing partial analysis results together with their dependency information. The remaining key to successful incrementality has been to relentlessly pursue local decision making. My analyses create dependencies locally, during the ordinary course of their processing. The lazy analysis described in Sections 6.4.3 and 6.4.3 uses local call site information to decide whether library code is referenced and local declaration-site information to detect whether that code has already been processed. More generally, *all* incremental behavior in my analyses happens as the result of simple, local decisions.

6.5.2 What to traverse

In this section I discuss two different engineering trade-offs, each of which centers around the questions "What view of the program code should I traverse?" and "How efficient will that traversal be?" The first trade-off involves my decision to switch the core thread coloring analysis from my initial implementation, a flow-based analysis that traverses the control-flow graph (CFG) inside each method, to a syntax-based analysis that traverses the Fluid AST (fAST). My goals in making this change were to speed up the thread coloring analysis and to simplify the thread coloring language that is presented to users. It was wildly successful on both fronts. The second trade-off involves my decision to recode my analysis so that it traverses a slimmed-down version of the fAST containing only the constructs of interest to thread coloring analysis, as seen in the current version of that analysis. My goals in making this change were to simplify the implementation of the analysis, and to speed up the analysis yet again. Experience shows that I achieved the desired simplification, but that the new implementation is somewhat slower than when I used the fAST directly. This second trade-off was neither a success nor a failure.

CFG or AST?

To discuss the tradeoff between traversing the CFG and the fAST, I first set the stage with a brief description of the thread coloring language and the thread coloring analysis as they then stood.

The original thread coloring language Recall from Chapter 2 that the thread coloring language consists of more than just the annotations. It also includes both the possible program points at which annotations may be placed and also the meaning of those placed annotations. In the original thread coloring language, each `@Grant` annotation—which could appear at any program point—was treated as a definition point; specifically, it modified the color binding of the current thread by adding a color.[13] Similarly, each `@Revoke` annotation—which could also be placed at any program point—was also a definition point, modifying the color binding of the current thread by removing a color. There was no requirement that each `@Grant` have

[13]I did not yet support granting or revoking lists of colors in a single annotation.

a single—or indeed, *any*—matching `@Revoke` annotation; it was entirely possible to grant a color inside a method and then return with a color binding that had changed during execution of the method. A key implication of these properties of the thread coloring language is that the analysis implementation must be flow sensitive.

The original thread coloring analysis As discussed in Chapter 5, the current implementation of the thread coloring analysis operates in terms of "color environments," each of which represents the set of all possible color bindings that may execute at a particular program point. In this section I describe an earlier stage in my research, when I had not yet developed the idea of color environments. At that time, the thread coloring analysis operated in terms of "*the* color binding" of the current thread; in this implementation each color was either certainly bound to the current thread, certainly not bound to the current thread, or was a "don't care" case. This original thread coloring analysis implementation supported Boolean expressions built using the AND and NOT operators, but was unable to support expressions built using the OR operator. The original implementation also assumed the presence of an explicit color constraint for every method-like entity in the program.

The color binding was represented as a pair of sets, one to hold the known-bound colors and a second to hold the known-unbound colors. In combination, these sets allowed me to represent the binding state of the relevant portion of the possibly infinitely-sized space of all color names. As in the current implementation, I used the explicit color constraint for each method to form the initial state of the color binding.

The original implementation of the thread coloring analysis proceeded by modeling the changes to the current color binding within each method. The analysis was modeled after a standard value propagation analysis, that is, it was a forwards flow-based analysis.[14] As is typical of such analyses, it required iteration to a fixed point.[15] This iteration required at least $\mathrm{O}\left(ld+1\right)$ iterations, where ld is the static loop nesting depth of the method being analyzed. At this stage in my research, I had not yet added annotation inference to the system.

Problems with the original language and analysis The original thread coloring language and analysis were unsatisfactory in many ways; most importantly, their focus on "*the* color binding" was fundamentally flawed. The possible color bindings—now known as the color environment—can represent real programs; the original implementation could not. This crucial flaw, however, was discovered and repaired separately; it is not the focus of this section. The problems of interest in this section are:

Developer understanding I found it nearly impossible to understand why a particular set of annotations led to an inconsistent (or consistant) result. This problem arose because the original analysis presented its results with no indication of the "reasoning" behind them. I solved this problem by adding to the analysis algorithm a set holding references to the specific annotations that were the source of each color name present in the two preexisting sets. This deponent set provided a ready-made source of information to help me understand the reasoning behind the results my tool was producing. More importantly, it taught me how to track deponent information through an iterative flow-based analysis. This lesson enabled my semi-incremental color inference algorithm in the current implementation of the prototype assurance tool.

End-user understanding Even after adding the computation of deponents, I found it nearly impossible to explain to end users the proper placement of `@Grant` and `@Revoke` annotations. The relatively precise statements that "You should grant a color at a program point that dominates all uses of that color" and "You should revoke the color at a program point that post-dominates all uses of that color" are utterly meaningless to most programmers. And more colloquial statements of these rules are both complicated and confusing. This approach was a complete failure in terms of adoptability.

Annotation volume The assumption that every method would have an explicit color constraint was badly flawed. The number of annotations required was too large; end users bluntly refused to write so many

[14]There was an additional backwards analysis pass using the requirements of invoked methods; this backwards pass produced suggestions to assist in writing constraints that were consistent with the constraints on invoked methods. This pass suffered all the same usability and performance issues described for the forwards analysis, and so will not be discussed farther.

[15]Astute readers will already have noticed that lack of support for Boolean-OR caused incorrect behavior at CFG join-points. My later realization of this problem led me to the idea of color environments and use of Boolean expressions in place of sets.

annotations. Even *I* was not willing to write that many. This failure led me to the idea of annotation inference which, in turn, added yet another layer of iterative analysis as described in Section 6.4.3 on page 150. Adding another layer of iteration however, led immediately to issues of poor performance.

Performance With the iterative color inference algorithm added to the prototype assurance tool, thread coloring analysis runs became extremely slow. Having an iterative flow-analysis nested inside the iterative color inference analysis, all of which then ran over entire programs was clearly not a recipe for success.

Around this time, my colleagues in the Fluid project reported the results of their performance analysis of the Greenhouse–Boyland effects and Greenhouse locking analyses. In a nutshell, they reported that flow-based analyses using the IR's CFG were extremely slow,[16] and that by comparison syntax-tree based analyses were effectively instantaneous (my paraphrase). At the same meeting, I reported my frustration with the extremely poor performance I was seeing in my thread coloring analysis. A brief brainstorming session produced the suggestion "Why not try a syntax-based analysis for coloring within methods?" The obvious objection was that a syntax-based approach would be unable to analyze some patterns that the original flow-based analysis handles easily; the syntax-based analysis is strictly less powerful than the flow-based analysis. Nevertheless, I chose to experiment with the AST-based approach.

My experimentation centered around two issues: language expressiveness and analysis performance. As seen in Chapters 2 and 5, the AST-based approach permits `@Grant` and `@Revoke` annotations only at block boundaries; the effect of these annotations "expires" on outbound block exit. The syntax-based approach cannot handle examples such as:

```
{
  if (cond) {
    // @Grant a
    ...
    // OLD thread coloring language, so color a DOES NOT expire here.
  } else {
    // @Grant b
    ...
    // OLD thread coloring language, so color b DOES NOT expire here.
  }
  // OLD thread coloring language, so
  // the Color environment now includes the clause (a | b)
  ...
  // @Revoke a, b
  // colors a and b are both gone now
  ...
}
```

In this example we grant different colors in the two arms of an **`if`**-statement, allow those colors to live beyond the **`if`**, and then revoke both colors sometime later. There are, of course, many other cases where a CFG-based analysis is more powerful than a syntax-based approach.

Rather to my surprise, I have never yet encountered an example for which I have needed the added expressive power of the original thread coloring language and its supporting CFG-based analysis. I have occasionally needed to insert extra blocks solely to hold a `@Grant` or `@Revoke` annotation; *I have never needed to restructure control flow to allow the syntax-based approach to succeed.*

The second issue was analysis performance. How large a difference would I need to justify the loss of expressive power described above? The resulting performance improvement was everything I had hoped for. Measurements at the time showed a performance improvement ranging from fifteen times faster than the CFG-based analysis up to fifty times faster (15x–50x).[17] The improvement was so large that analysis runs on small examples required seconds instead of minutes; analysis runs on large examples took minutes, rather than hours. This was a very convincing performance improvement; I would be extremely reluctant to give up such an impressive gain.

A third issue that I should have considered, but did not, arose during the Fluid group's next case study engagement in the field. After mastering the basic idea of thread coloring, end users had no problem understanding the rule: "`@Grant` and `@Revoke` annotations may appear only at block boundaries; the effect of

[16]Subsequent engineering has improved this situation; such analyses are now merely "somewhat on the slow side" rather than the previous "glacially slow."

[17]These gains would be much smaller now that the IR-based CFG has been re-engineered, perhaps in the range of 2x–5x.

these annotations "expires" on outbound block exit." This simplification of the thread coloring language *also* solved my problem with end-user understanding.

Enduring Lessons

- Expressive power does not matter if the users do not understand it.
- Trading expressive power for performance is sometimes a good idea. When the expressive power is not crucial, choose performance instead.
- The question "Why is this so?" is crucially important for understanding analysis results. However, an approachable presentation is also important.
- Reaching-deponents is a simple way to track dependencies through iterative analyses; this technique computes the data needed to understand *why* some result is true.

AST or auxilliary slimmed-down AST?

More recently, during a case study experience in the field, I was asked to analyze a 200KSLOC chunk of a *much* larger program. I was dismayed to find that each analysis run was slow enough to encompass a brief bull-session with the end-user developer I was assisting. It is important to note that the module and thread coloring analysis algorithms were essentially in their current form and that the analyses were implemented incrementally as described above. On my return to CMU, I re-visited the Electric case study, partly to see whether my analyses could process all 140KSLOC in a single chunk, and partly to address performance issues.

During this study I noticed that my analyses only cared about a few kinds of AST nodes, and that my processing for many of those nodes was identical. Specifically, thread coloring analysis cares only about the following node kinds: CUs; Types (*e.g.,* classes, enums, Interfaces, anonymous nested classes, *etc.*); declarations of method-like entities; blocks; invocations of method-like entities; and the subset of data references that touch data regions for which we are computing color information.[18] Furthermore, each of these six groups of node kinds invokes identical processing for every AST node that fits in that group. I realized that this observation offered the opportunity both to refactor my code so it would be easier to understand and also to improve performance by reducing the number of nodes traversed on each analysis pass (while also traversing simple Java objects rather than IR-based trees).

Refactoring I promptly performed the obvious refactoring of my code to reflect the commonality of processing. This change was extremely successful. I removed large amounts of redundant code; the surviving code base was greatly simplified and was much easier to understand.

Slimming the AST I implemented a new "color tree" structure consisting of the six node kinds listed above. Those classes that permit children use Java `Collections` to hold their children, typically of type `List` with the actual object being an `ArrayList`. To avoid repeated IR lookups, each color tree subclass has fields that hold the data needed during thread coloring analysis; these fields are initialized once when the node is built. The color tree nodes also maintain a link back to the original IR node in whose place they stand. The nodes themselves are ordinary Java objects; they are not stored in the IR. These "color trees" are built on a per-CU basis as needed just prior to the first thread color analysis pass. I modified all of the subsequent AST traversal code in the thread color analysis passes to traverse the newly created "color tree" structure instead of the IR's AST.

The color tree structure is significantly smaller than the AST in terms of number of nodes. It excludes all expression operators, all control flow, all local variable references, and all field references except those that involve data regions of interest to thread coloring analysis. I confidently expected that access to the data needed during the analysis would be much faster, because access to IR attributes always requires at least one lookup in and indirection through a map, and occasionally more; by comparison the equivalent accesses in the color tree are simple field accesses for data about an individual AST node, or simple `ArrayList` traversals for iteration over children.

[18]Recall that all constructors are explicitly present in the IR.

Performance Result Once the changes were complete and debugged, I eagerly timed analysis of Electric. My new code was 1.4x *slower* than the previous version. It was clearly time to investigate more carefully.

Closer analysis showed that my AST traversals were indeed faster than in the prior version—about 1.3x faster. The slow-down originated in two places. First, the time to allocate and build the color tree was itself larger than the amount of time saved in a single analysis run. Secondly, perhaps due to the additional storage allocation, I saw many additional garbage collections. In total, these extra garbage collections consumed about as much time as building the color tree. It may be relevant to note that, when processing all of Electric v8.0.1 in one chunk, the prototype assurance tool is on the verge of running out of memory in a 32-bit JVM. I have not measured garbage collection cost for smaller examples.

Because thread coloring analysis rebuilds ASTs only for CUs that change, however, end users get the benefit of faster traversals on subsequent re-analysis passes for those units that both did not change and that required additional processing in spite of the incremental analysis. My experimentation suggests that the break-even point comes somewhere between the third and fourth re-analysis run. This will, of course, depend on the details of both how many and which specific CUs are modified.

Enduring Lessons

- Refactoring for clarity is a good idea.
- Performance, however, is complicated; it may be a mistake to refactor for performance reasons. In this particular case, my original analysis was correct about access and traversal costs. However, I omitted the cost of building the auxilliary tree, which turned out to be significant.

6.6 Usability Issues

Addressing issues of usability and practicability consumed the vast majority of my work on this system. Earlier sections have addressed many of the practicability issues through discussion of performance, incrementality, and the like. In this section, I discuss some of the challenges I encountered in attempting to make my analyses usable by ordinary end users. In each section I describe the solutions deployed to date, along with ideas for further improvement.

6.6.1 Reporting

Attempting to report results in a usable fashion brought significant challenges. I now discuss some of these challenges along with partial solutions, followed by a brief description of a number of unresolved issues.

Volume of messages

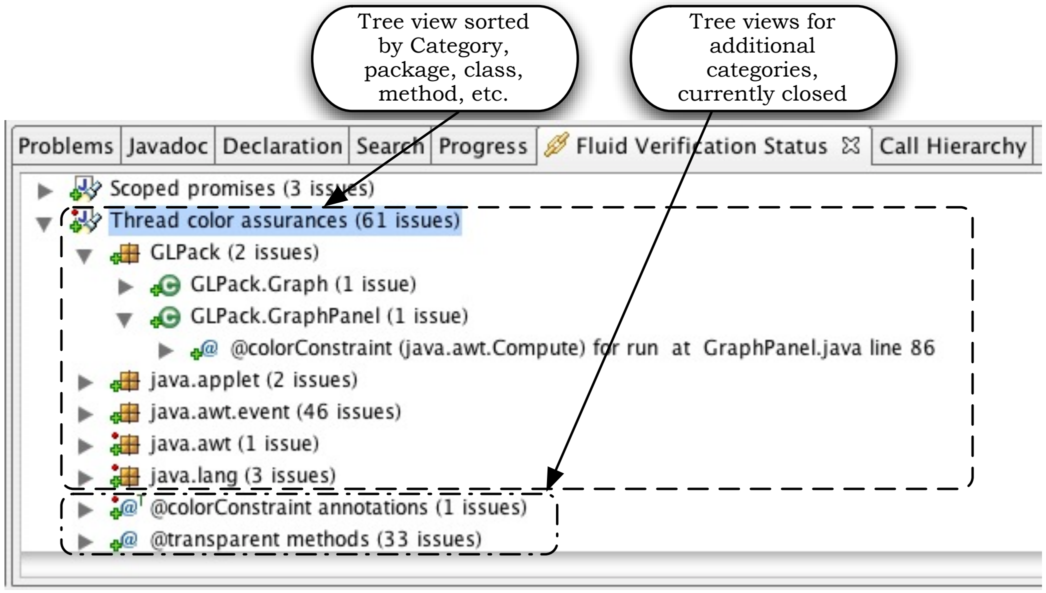
As previously discussed in Section 6.4.3, thread coloring analysis produces a very large number of messages. The Fluid group has applied several approaches to assist users in making use of these messages.

First, messages are sorted and folderized in the Fluid Verification Status window. That is, all messages from a single method are grouped together into a folder in the tree view. The folders for all the methods in a class are similarly grouped, as are those for all classes in a package. Items within any given folder are lexically sorted. This simplifies finding the messages pertaining to code of interest to the end user.

Secondly, messages are categorized. That is, color errors are grouped together, as are color warnings, *etc.* Figure 6.3 on the facing page shows the Fluid Verification Status window with overlays identifying sorted messages. The highlighted line reading "Thread color assurances (61 issues)" is a top-level category, as is the line for "Scoped promises."

Finally, each analysis can request placing a particular category of messages at the top level of the report. For thread coloring, I have chosen to request top-level placement for reports of coloring inconsistencies. In this fashion a user can go directly to the errors for any particular piece of code, without needing to sort through the non-error messages.

These techniques help end users find the messages they are interested in, but do not help reduce the time it takes Eclipse to present the Fluid Verification Status window when the number of results is very large. For


Tree view sorted by Category, package, class, method, etc.
Tree views for additional categories, currently closed
Problems
Javadoc
Declaration
Search
Progress
Fluid Verification Status
Call Hierarchy

```
▶ Scoped promises (3 issues)
▼ Thread color assurances (61 issues)
  ▼ GLPack (2 issues)
    ▶ GLPack.Graph (1 issue)
    ▼ GLPack.GraphPanel (1 issue)
      ▶ @colorConstraint (java.awt.Compute) for run  at  GraphPanel.java line 86
  ▶ java.applet (2 issues)
  ▶ java.awt.event (46 issues)
  ▶ java.awt (1 issue)
  ▶ java.lang (3 issues)
  ▶ @colorConstraint annotations (1 issues)
  ▶ @transparent methods (33 issues)
```

Figure 6.3: Fluid Verification Status window

the Electric case study, presenting the Fluid Verification Status window takes more time than the entire thread coloring analysis does. Clearly, some optimization is in order here.

Displaying Large Boolean Expressions

The messages for many thread coloring analysis results include the Boolean expressions that represent color environments and constraints. Thread coloring analysis operates on fully renamed expressions with fully bound names and with the effect of `@IncompatibleColors` made explicit. That is, an apparently small user-written expression such as `(AWT | IPLocator)` may actually be represented as

```
((java.awt.AWT & !java.awt.Compute & !org.shetline.skyviewcafe.IPLocator
    ) | (!java.awt.AWT & java.awt.Compute & org.shetline.skyviewcafe.
    IPLocator))
```

The actual expression from the SkyViewCafe case study is much larger, as there are nineteen additional colors that are mutually incompatible beyond `AWT` and `IPLocator`. Even without using fully qualified names, these expressions get large very quickly. It is not uncommon for the string representation of an expression to be longer than Eclipse's maximum string length for a message in a tree-view. Clearly, the prototype assurance tool needs a way to take a Boolean expression and present the original simple expression written by the user.

Sadly, processing an arbitrary Boolean expression to detect those portions of it that are attributable to a particular renaming or incompatibility expression and then unwinding to the simpler version turns out to be a hard problem. I wrestled with it for several weeks without making noticeable progress.

Fortunately, there is a much simpler approach based on an important property of Binary Decision Diagrams (BDDs). For any specific Boolean expression, there is exactly one BDD that represents that expression; thus the BDD can be used as the key to look up a saved representation of the expression. While converting user-written expressions into fully renamed BDDs with all incompatibilities included, the thread coloring analysis support code maintains a map from the BDD to a string holding the shortest (by character count) user-written expression that produces that BDD. When formatting BDDs for analysis results, the support code uses the BDD as the key to look up the saved expression. Thus, even though the "real" version of the example expression above contains two clauses each with twenty-two terms, the user-visible message contains the string `(AWT | IPLocator)` instead.

When a BDD does not match any user-written expression, the support code falls back on printing the expression in Disjunctive Normal Form (DNF). To shorten the printout somewhat, it prints fully qualified names only when ambiguity is possible; it uses the short name for color names defined in only one place.

This approach to displaying large Boolean expressions is usually successful. However, it fails in two scenarios. The obvious failure comes when there is no user-written expression that produces some particular Boolean expression that is both common in the results and large when printed. The current work-around for this failure mode is to write a `@ColorRename` expression whose right-hand-side evaluates to the expression of interest and whose left-hand-side is a distinctive short name. This has the benefit of fixing the problem for that expression. However, the need for additional user action is unfortunate.

The less obvious failure appears when there are multiple user-written expressions producing the same underlying Boolean expression. The display code defaults to printing the lexically shortest name for the expression. This is occasionally surprising when the most common user-written version of the expression is slightly longer than an uncommon and perhaps poorly chosen renaming. For example, suppose that the example expression `(AWT | IPLocator)` appears with this "spelling" in nearly all cases; suppose further that the user then adds a renaming declaration such as "`@ColorRename fred **for** (AWT | IPLocator )`". The result will be that all Boolean expressions that were previously printed as "`(AWT | IPLocator)`" now appear in the Fluid Verification Status window as `fred`. This seems most unfortunate.

Reporting the genesis of the current color environment

When examining color errors, it is very difficult to tell *why* a particular message is present. The messages include supporting information that indicates the specific failure, of the form "Current color environment (AWT) does not imply requirement (Compute)" for example. This information is often enough to resolve the issue, typically when the programmer expected the call site to be executing in a color environment that included the AWT color. In this case, the bug is probably a mistaken call. Conversely, when the invocation and the constraint on the invoked method are as expected, the issue is that the current color environment is unexpected. To resolve this problem, the user needs to know where that environment came from. But the Fluid Verification Status window's tree view shows only dependent results, not deponents. And the simple fix of inverting the tree to show deponents and not dependents gains the user nothing, because there are a similar number of cases where the dependents are what the user needs to see. The information is available in the TMS; the view does not present it.

This problem is especially vexing when the unexpected color environment is "true." This condition arises when there are one or more methods belonging to some module's visible interface (alternatively API) that both do not yet have a declared color constraint and transitively invoke the method where the failure is reported. Such methods have the least-knowledge default constraint—"true." Because the join operation in the color inference algorithm is Boolean-OR, this "true" propagates downwards through the call graph as the inferred constraint (and thus the color environment) for all unannotated methods transitively invoked from the unannotated API method. Finding the unannotated method or methods currently requires tedious use of Eclipse's call hierarchy and searching capabilities.

Making the deponent information from the TMS available to the user may not be a sufficient improvement to achieve adequate usability. The deponent information would significantly reduce the search space, but the transitivity property may still make it difficult to find the unannotated API methods. Fortunately, a property of BDDs allows an inexpensive improvement in reporting.

The property of interest is this: BDD variables are cheap. Specifically, variable substitution has near-zero cost as long as the size of the Boolean expressions being evaluated does not increase. And even if the size of some Boolean expressions does increase, the cost increase is only linear in the number of variables added.

Armed with this property, the analysis code can make the Boolean expressions themselves indicate which specific user-written annotations were the original source of the expression being examined. The technique works as follows: at each user-written annotation, the analysis code replaces the named colors with newly created derived variables whose identity carries with them the location of the annotation as well as the named color. Thus, for example, the annotation `@Color AWT` at line 123 of file foo.java might produce the expression `(AWT.foo.java_line123 & !Compute.foo.java_line123)`. By keeping a map from introduced variable to the original color name, the analysis code can treat these variables as being exactly equivalent to their original for semantic purposes, while still letting their specific identity indicate which particular annotation or annotations produced them. By introducing a special out-of-band color name, say `_unknown` , and using it for all unannotated API methods the analysis can achieve the same effect in place of the current **`true`**. This technique would allow immediate access from any computed color expression back to the annotations that engendered it.

This would, of course, require enhancement of the reporting infrastructure to present the information to the user on request, and otherwise behave as it does now.

Other reporting issues

The Fluid Verification Status window displays all results as a tree with dependent results as children in the tree. While performing thread coloring, end users often wish to know about both dependents and deponents, but the tree view can show only one link. Furthermore, thread coloring results are actually a graph, not a tree. This mismatch makes performing thread coloring analysis more difficult than it needs to be. A better solution may be to investigate research on presenting graph-structured results.

The wording of thread coloring analysis messages is not always clear. Improved wording for the messages would assist end user understanding.

6.6.2 Writing annotations

In previous sections I have discussed several techniques I have used to reduce the number of annotations that an end user must write to divide her code into modules and then color her code and data. Writing annotations remains tedious even after application of these techniques. I next discuss several possible improvements that could make writing annotations still easier. These possible improvements remain unimplemented in the interest of timely completion of my thesis.

Proposed annotations

The module analysis discovers every cross-module reference; it further identifies those that correctly obey the visibility rules and those that are in error. My experience with existing bodies of code shows that when first modularizing a program, most cross-module references are in fact correct; the problem is that the entity to which the code refers has not yet been marked visible. A reasonable heuristic may be to propose all such entities as candidates for visibility. The module analysis even knows how far up the module hierarchy any particular entity should be made visible, and could propose this along with the rest.

As described previously, thread coloring analysis requires a color annotation for all entities that are part of the visible interface (sometimes called the API) of their home module. The analysis code knows which entities require annotation; it could provide the user with a list, or perhaps an Eclipse search window prepopulated with these entities. Existing code that is only partially colored often contains calls from code that has a known color environment to API methods that lack a color constraint. My experience with analysis suggests that most, but not all, such API methods will eventually be colored with a constraint that is quite similar to, or even identical to, the calling color environment. An enhanced assurance tool could propose the observed coloring as a possible choice to the user.

It is quite important that the assurance tool not apply these choices automatically, however. It can use the observation that *most* code obeys the intended module structure and thread usage to make the end user's life easier. That said, it must not obscure the few instances that are actually wrong. The intent here is to offer the user a short-cut, but to remind her that the *correct* choice depends on her intent. The short-cut is only a guess.

"Make it so" button

In concert with proposed annotations, the enhanced assurance tool could provide a "make it so" button or a wizard to assist the user with the mechanics of actually installing the proposed annotations. The interface might use a check-list for approving or rejecting individual choices, along with an action button to request insertion of the approved annotations at the approved places. The enhanced tool might use Eclipse's refactoring support to change the source code appropriately.

6.7 Future Work

6.7.1 Getting to prime time

I now discuss the items that must be addressed to take this prototype implementation of my module and thread coloring analyses from their current research status to that of a commercial quality implementation.

New Features This dissertation describes a number of new features that would enhance the expressiveness of the thread coloring language and so make the prototype assurance tool easier to use. A commercial-quality tool should include color parameters for variables, types, and method arguments (Section 5.5.2); and color inheritance (Section 5.5.1). Further, we should upgrade the tool to recognize which accessing color environments indicate that the accessed data region is actually confined to a single thread; this reasoning is currently left to the user.

Practicability The prototype assurance tool performs adequately in most cases. Nevertheless, a commercial quality tool should perform well even on extreme examples. To this end, it would be necessary to investigate and resolve the performance issues with reporting many thousands of results as well as to investigate techniques for managing and presenting large numbers of reported items, including graph-structured results. Further, we should investigate and implement a tasteful selection of incrementality improvements for both the analysis algorithms and the result reporting code (see Section 6.5.1).

Usability Commercial customers would not tolerate the reporting issues described earlier in this chapter (Section 6.6 on page 160), most notably the lack of deponent information and the difficulties finding the origin of unexpected color environments. Any commercial tool must resolve this problem. Implementing wizards and proposed annotations would greatly improve the user experience, and should be considered for any commercial implementation.

Reliability The prototype assurance tool is just that: a prototype. A commercial-quality tool must achieve significantly better reliability. This would require both reimplementation of some of the code and significant debugging.

6.7.2 Other future work

We could improve both the precision and the performance of the thread coloring analysis by pruning the call graph using well-known optimizations that apply reaching definitions to refine information about the types of the objects whose methods are being invoked. Similarly, it would be interesting to investigate whether clever hashing of book-keeping data structures can reduce their cost from O(lg n) to constant cost per access and whether such an effort would be worthwhile in terms of performance improvement.

The module analysis as currently written supports only those annotations and features that were required for this thread coloring work. The additional features as yet unimplemented include those most likely to interest developers of large systems. We should implement and experiment with these features.

This thesis includes only a back-of-the-envelope level analysis of the asymptotic performance of thread coloring analysis. A rigorous performance analysis of both the as-built tool and also the intrinsic asymptotic performance of the analysis would provide guidance to future implementors.

6.8 Conclusion

The prototype implementation of thread coloring has been surprisingly successful. A key enabler of that success was the early decision to base the tool on the Fluid group's infrastructure. Key benefits provided by that infrastructure include:

- Complete Java front end support—Fluid provided a complete AST, binder, and all other necessary Java infrastructure. This was originally all home-grown; portions were later replaced as part of Eclipse integration.
- The Drop–Sea truth maintenance system by Halloran [29], along with its canonical use in the Fluid system, enabled my experiments with incremental recomputation of results.

- The Greenhouse–Boyland effects analysis provided thread coloring's connection to data. Without this capability, thread coloring would have addressed threads and code with no connection to data.
- Fluid's annotation infrastructure greatly simplified defining new annotations and linking them with the appropriate nodes in the AST. Thread coloring began long before Java 5 annotations; without the Fluid annotation infrastructure, experimentation would have been much more difficult. The Fluid annotation infrastructure allows placement of annotations in more places in the AST than does Java 5; this flexibility is crucial for thread coloring.

Incremental recomputation provides a way to reduce work, thus providing significant performance benefits. The common wisdom in the static analysis community is that incremental recomputation of results is too difficult and error-prone to be worth the effort. A surprising result of this implementation effort is that incremental recomputation can be easy. The two key factors are the availability of the necessary infrastructure—most notably a fine-grained dependency management system—and a careful focus on making invalidation and recomputation decisions local. The Drop–Sea TMS provided the infrastructure and inspired my attempt to build incremental analyses. The focus on local decision making was a natural consequence of the TMS's style of invalidating fine-grained results. Armed with this support and approach, I have not encountered any incrementality bugs.

Trading expressive power for performance is sometimes a good idea. In particular, expressive power does not matter if the end users cannot make use of it because they do not understand it. For example, the original flow-based thread coloring analysis was strictly more powerful than the current syntax-based analysis. However, the expressive power of the flow-based version was wasted, because it confused the end users; they were unable to take advantage of it. Not only did switching to a syntax-based approach remove the cause of end-user confusion, it also improved performance dramatically.

The question "Why is this so?" is crucially important for understanding analysis results. Users of static analysis tools encounter this question every time they see an unexpected result. Computing reaching-deponents (see Section 6.5.2) is a simple way to track dependencies through iterative flow-based analyses; reaching-deponents provides exactly the data needed to understand *why* some result is true.

The traditional criteria for success in static analysis research are soundness and scale. Researchers typically invent an analysis, demonstrate soundness, evaluate its scaling properties, and publish. In this research, we addressed the additional criteria of practicability[19] and adoptability. Evaluating the practicability and adoptability of thread coloring analysis required addressing a variety of hard problems that would benefit from future research, including managing and presenting large numbers of results, supporting incremental adoption of an analysis technique for large bodies of existing code, and other ease-of-use issues. Addressing practicability and scale for other analyses would likely lead to other hard problems, and open still more interesting areas for additional research.

[19]Note that addressing practicability is *not* the same as "engineering a commercial-grade solution." Rather, practicability is the demonstration that the analysis in question can be made to be practical.

Chapter 7

Conclusion

7.1 The Problem

Concurrent programming has proven to be difficult. Concurrency issues are diverse; they include thread confinement, thread roles, deadlock, liveness, atomicity, and especially state consistency. One widely used approach to state consistency is the use of locks—both lexically scoped and dynamic. This is known as "synchronization" in Java, and is widely studied [1, 2, 5, 33, 12, 15, 23, 27, 30, 42, 49, 57]. However, developers may find it difficult to understand which data regions may potentially be shared, as we saw in Sections 4.3 and 4.7.

Another, and often more appropriate, approach is through the use of policy rules that specify which threads may perform which actions. We call this approach "non-lock concurrency" and call the set of rules a *thread usage policy.* These rules may mandate thread confinement, restrictions on which methods may be invoked from which threads, or restrictions on which threads may touch which shared state.

This non-lock concurrency approach is not language-specific; for example, it appears in the design of the Windows GUI [43] regardless of programming language. Many libraries and frameworks use this approach to concurrency. Modern graphical user interfaces are perhaps the best-known example, but thread usage policies also appear in the JavaBeans framework, as well as in many individual applications.

As currently used, these policy-based approaches encounter a variety of difficulties. The thread usage policy associated with a framework is often not well documented, difficult to find, or unstated. We found preexisting explicitly stated thread usage policies in only two of our fifteen case studies (see Sections 4.5 and 4.8). The documentation is also often out of date or incorrect; neither of the preexisting policies we encountered in the field was both accurate and complete. This may be due to the fact that thread usage policies are also difficult to express; the typical approach today is to write prose. But prose often fails to state precisely the intended policy, and cannot be checked for consistency with as-written code.

Thread usage policies may be hard to follow, because the implications of a policy apply to widely scattered pieces of code (see Section 4.5). Furthermore, a thread usage policy is often not easily evident from examples or client code. This difficulty arises partially due to difficulty distinguishing essential features of the example code from accidental features, and partly because there is no statement in the code that indicates that a particular pattern is used in order to comply with the thread usage policy. When a thread usage policy is unstated, it may be difficult or impossible to reverse engineer from source code; one of our laboratory case studies ended due to this difficulty (see Section 4.6).

Finally, violations of these policy rules are hard to diagnose. The symptoms of a policy violation may be difficult to recognize as being related to thread usage policy problems (see, for example, [47]). Even if this step of diagnosis is made, it may be difficult to find the specific point in the source code at which the policy was violated.

7.2 The Vision

Our idea is to introduce a formal language to document thread usage policy in source code. The notation is precise enough for automatic checking, yet familiar enough for programmers. We also introduce a static analysis that supports semi-automatic checking of thread usage policy and as-written code for mutual consistency. Finally, we introduce a new combination of preexisting techniques that reduce policy-expression effort to a level that is plausibly acceptable to programmers.

7.3 Hypotheses and Requirements

In this thesis we introduce thread coloring, a framework of discourse intended to support expression of and reasoning about intended thread usage policies for a wide variety of code in a practicable and scalable way.

We had four principal hypotheses:

- **The thread coloring language can provide a useful and concise expression of thread usage policies.** We have developed a group of annotations that concisely express thread usage policies. They are described in Chapter 2.

  We have defined the static semantics of thread coloring. This definition allows us to reason about the properties of thread coloring; it is presented in Chapter 5. Because the focus of the thesis is on practicability issues, we deferred to later work meta-theoretic proofs of the analysis itself.

  To evaluate the usefulness of the thread coloring language, we modeled thread usage policies in case studies involving a wide variety of code and of policies. We then assessed whether the models were useful (by observation) and whether developers could understand them (using informal reports from professional developers).

- **In particular, we hypothesize that thread coloring can help address a range of concurrency issues—such as assuring single-thread access, identifying possibly-shared data regions and localizing knowledge about roles for threads—that have not previously been addressed in a comprehensive and systematic fashion.** Thread coloring has broad utility. We have performed and evaluated case studies, both in the laboratory and in the field, on a broad selection of code from a variety of application domains. The six individual case studies listed in Table 4.1 are discussed in greater detail in Chapter 4. They involved a broad range of scale, application type, and thread usage policy models. The case studies presented include a small example program (see Section 4.3) and a 140KSLOC VLSI design tool (see Section 4.5); applications built using three different GUI toolkits and an action-planning and optimization application that has no GUI at all (see Section 4.8); a graphical editor (see Section 4.6) and an astronomy application (see Section 4.4), among others. These case studies came from various open source projects and from fielded applications both commercial and scientific.

  Laboratory case studies were selected to explore specific aspects of thread coloring, as indicated in Table 4.1. Field case studies, however, were *not* pre-selected. Rather, the studies were set up with organizations whose developers were encountering problems that our analyses might address. The selection of case studies was done by the outside organizations; we saw the code for the first time on arrival for the case studies. The Electric case study was unusual in that the sponsoring organization is working with an open-source artifact. Thus, we were able both to begin work in the laboratory before the in-field case study, and also to continue working in the laboratory after returning from the in-field case study.

  Thread coloring helped address a range of concurrency issues. The case studies showed how thread coloring addressed the following concurrency issues and assurances:

  - Methods invoked in the "right thread," more properly "correct thread role" (all case studies, but see especially Section 4.3)
  - Data regions touched only by the "correct thread roles" (see Section 4.7)
  - Detection of potentially-shared data regions (see Section 4.3). These data regions are then candidates for locking analysis, which is heavily studied in related work, such as [23]

- What thread roles should "get here?" (all case studies, but see especially Section 4.3)
- What thread roles actually may "get here?" (all case studies, but see especially Section 4.3)
- What threads (or thread roles) are relevant? (all case studies, but see especially Section 4.3)
- Localizing knowledge about roles for threads, by propagating the thread colors through the as-written source code. (all case studies, but see especially Section 4.3)
- Assurance that data regions intended to have only single-thread access are in fact accessed from only one thread. (see Section 4.7)

- **This thesis further hypothesizes that it is possible to build a tool based on thread coloring that realizes a useful and practicable technique for assuring consistency between the expressed model and the as-written code.** We have built a prototype tool as an existence proof. Chapter 6 discusses its implementation and some of the lessons we have learned while building it and experimenting with it. Specifically, we undertook a number of case studies which are detailed in Chapter 4. The prototype assurance tool found inconsistent thread usage in widely used programs, both commercial and open source, and provided thread usage policy assurances that were not otherwise available. Anecdotal evidence from developers suggested that they highly valued these results.

  We have demonstrated successful results using data coloring (see the JPL Chill GDS case study, Section 4.7 for details). In that in-the-field case study, the developers were primarily interested in determining whether certain data regions of interest were confined to the SWT event thread. Once the tool showed this through analysis, the developers removed all synchronization from the data regions of interest, achieved much of the performance increase they sought, *and* passed their regression tests. They commented that without the tool's analysis they would not have been willing to remove synchronization, for fear of introducing a race condition or other threading-related bug.

- **This thesis hypothesizes that with this tool, it is possible to reduce the effort required for annotation and analysis sufficiently both to use thread coloring on larger bodies of code, and to plausibly enable adoption by practicing programmers.** We evaluated scalability, adoptability, and general practicability by evaluating a variety of criteria during our case studies. We measured model expression effort in terms of annotations per thousand lines of code (annotations per KSLOC). The experiment described in Section 3.5.2 shows a current measure of 6.3 annotations per KSLOC; we believe that this can be reduced in future by an additional order of magnitude.

  Scalability to very large programs, however, requires separable composable analysis. To support separate analysis of parts of large programs, we designed and implemented a module system for Java (see Chapter 3). The prototype assurance tool can divide large programs along module boundaries, processing each module individually. Because thread coloring annotations serve as analysis cut-points, the separate results can be combined to provide information about the larger system.

  We have performed a pair of scalability studies—on SkyViewCafe (a ~25KLOC shareware Java application; see Section 4.4) and on Electric (an ~140KSLOC open–source VLSI design tool; see Section 3.5). These scalability experiments suggest that the user-experience provided by adding thread coloring annotations and using the prototype tool is interestingly light-weight and that the algorithms used in the analysis have the potential to scale up to much larger applications. Algorithmic performance is discussed in Chapter 6.

  We experimented with work-flow issues, user-interface issues, support for reducing annotation and analysis effort, and assessed the actionability of the results produced by the prototype assurance tool during our in-field case studies. We evaluated developer comprehension and ease-of-use through informal reports from developers during our in-field case studies. Our incremental adoption model was a key ingredient in success with case studies in the field. Specifically, it enabled us to provide useful analysis results with partial annotation of smaller portions of large systems. The case studies show that thread coloring is a relatively straight-forward and manageable process even when one is working with other people's code.

7.4 Contributions to software engineering and lessons learned

This thesis provides five primary contributions to software engineering:

- It introduces a framework of discourse for thread roles, including the concept of thread usage policy, a specification language using annotations, and a semantic model of that language. Framework and library developers can use the specification language to express thread usage policies and to communicate those policies to the developers who are their clients. The language is familiar enough to satisfy ordinary programmers, yet formal enough to check automatically.
- It provides a systematic way to improve code quality by assuring that the as-written code is in compliance with the expressed thread usage policy. In particular, at each point at which the policy is relevant—that is constructor and method invocations, method bodies, and "interesting" data references—the tool either indicates compliance with the policy, or produces a specific error message showing the form of non-compliance.
- It demonstrates techniques to reduce model expression effort to very low levels. The techniques themselves are not new; the contribution is the demonstration that, when combined, the techniques provide a surprisingly large reduction in model expression effort. This reduction is enabled, in part, through the use of inference within modules. By keeping annotations succinct and sparse, these low levels of model expression effort greatly improve the developer's ROI, especially with respect to activities like writing specifications. Our belief is that this is key to making thread coloring an adoptable activity. This belief has been reinforced by developer's volunteered responses during our case studies, and by the experience of Microsoft and others with tightly targeted annotations.
- It demonstrates techniques whereby the consistency analysis can scale up to operate on very large programs—millions of lines of code appear to be within reach. These techniques include a module system for Java, which also supports separate development and separable composable analysis.
- It demonstrates techniques that permit incremental recomputation of results after a program change, and demonstrates that such incremental recomputation is straightforward. Incremental recomputation of results is not new; past work, however, has suggested that it is too complicated and error-prone to be worthwhile. This thesis contributes the lesson that with proper supporting infrastructure and a suitable approach, incremental recomputation is both straightforward and easy. The two key factors are the availability of the necessary infrastructure—most notably a fine-grained dependency management system—and a careful focus on making invalidation and recomputation decisions local. The Drop–Sea TMS provided the infrastructure, and inspired my attempt to build incremental analyses. The focus on local decision-making was a natural consequence of the TMS's style of invalidating fine-grained results. Armed with this support and approach, we have not encountered any incrementality bugs over the lifetime of this work.

Other lessons learned:

- Thread usage policies change over time. A prime example can be seen in the changing rules for thread usage in the AWT/Swing GUI frameworks. But many client developers are unaware of the changes in spite of substantial publicity and education efforts by the framework developers. Furthermore, client developers who are aware of the changes may find it difficult to understand the implications of the policy changes. Even very careful developers sometimes fail to understand these implications (see Section 4.4).
- Trading expressive power for performance is sometimes a good idea. In particular, expressive power does not matter if the end users cannot make use of it because they do not understand it. For example, the original flow-based thread coloring analysis was strictly more powerful than the current syntax-based analysis. However, the expressive power of the flow-based version was wasted, because it confused the end users; they were unable to take advantage of it. Not only did switching to a syntax-based approach remove the cause of end-user confusion, it also improved performance dramatically.
- The question "Why is this so?" is crucially important for understanding analysis results. Users of static analysis tools encounter this question every time they see an unexpected result. Computing reaching deponents (see Section 6.5.2) is a simple way to track dependencies through iterative flow-based analyses; reaching-deponents provides exactly the data needed to understand *why* some result is

true.

- The traditional criteria for success in static analysis research are soundness and scale. Researchers typically invent an analysis, demonstrate soundness, evaluate its scaling properties, and publish. In this research, we added the additional criteria of practicability[1] and adoptability. Evaluating the practicability and adoptability of thread coloring analysis required addressing a variety of hard problems that would benefit from future research, including managing and presenting large numbers of results, supporting incremental adoption of an analysis technique for large bodies of existing code, and other ease-of-use issues. Addressing practicability and scale for other analyses would likely lead to other hard problems, and open still more interesting areas for additional research.
- An approach that supports incremental adoption is necessary for practicability. This observation was reinforced during every case study; professional developers uniformly stated that they would not adopt tools that require either whole-program analysis or application to entire programs before producing interesting results. Developer feedback also indicated that very low model expression effort is *required* for adoptability.

7.5 Future Work

7.5.1 Thread coloring language

New Features This dissertation describes a number of new features that would enhance the expressiveness of the thread coloring language and so make the prototype assurance tool easier to use. A more powerful language should include color parameters for variables, types, and method arguments (Section 5.5.2); and color inheritance (Section 5.5.1). These features would enhance future field trials, and assist with adoptability.

Proof of Soundness A proof of soundness for the thread coloring analysis would add valuable rigor to our results. Such a proof requires the addition of a computation semantics for FJC. We could then prove that FJC's color consistency judgment is a sound extension of FJ. In addition, we could extend the semantics of FJC to support parameterized colors and color inheritance. This would then enable an expanded proof of soundness that would include those new language features in addition to the current features of FJC.

7.5.2 Assurance tool improvements

Practicability The prototype assurance tool performs adequately in most cases. Nevertheless, a more useful tool would perform well even on extreme examples. To this end, it would be necessary to investigate and resolve the performance issues with reporting many thousands of results as well as to investigate techniques for managing and presenting large numbers of reported items, including graph-structured results. Further, we should investigate and implement a tasteful selection of incrementality improvements for both the analysis algorithms and the result reporting code (see Section 6.5.1).

Usability The reporting issues described in Section 6.6, most notably the lack of deponent information and the difficulties finding the origin of unexpected color environments, are a major impediment to further field trials. Resolving this problem would greatly improve usability. Implementing wizards and proposed annotations would greatly improve the user experience, and should be considered for any future implementation. Further, we should upgrade the tool to recognize which accessing color environments indicate that the accessed data region is actually confined to a single thread; this reasoning is currently left to the user.

Reliability The prototype assurance tool is just that: a prototype. Improved reliability would expand the usefulness of the tool and enhance its performance in future field trials. This would require both reimplementation of some of the code and significant debugging.

[1]Note that addressing practicability is *not* the same as "engineering a commercial-grade solution." Rather, practicability is the demonstration that the analysis in question can be made to be practical.

Analysis Improvements We could improve both the precision and the performance of the thread coloring analysis by pruning the call graph using well-known optimizations that apply reaching definitions to refine information about the types of the objects whose methods are being invoked. Similarly, it would be interesting to investigate whether clever hashing of book-keeping data structures can reduce their cost from O(lg n) to constant cost per access and whether such an effort would be worthwhile in terms of performance improvement.

Rigorous performance analysis This thesis includes only a back-of-the-envelope level analysis of the asymptotic performance of thread coloring analysis. A rigorous performance analysis of the as-built tool as well as of the intrinsic asymptotic performance of the analysis would provide guidance to future implementors.

7.5.3 Modules

Check sealed module assumption To ensure the sealed module property needed for module-at-a-time inference, we should implement a trusted class loader for dynamically loaded code. This loader would check that the loaded classes are (a) the specific classes that were analyzed statically, (b) are being loaded "within the context" of the module that statically "owns" them, and (c) that the modules accessed/invoked via cross-module references are in fact the ones that we analyzed statically (or are sufficiently compatible that all analysis results will still hold). Property (a) is clearly feasible; properties (b) and (c) will require rather more work, but should be manageable with reasonable engineering effort.

Finish implementing designed features The prototype implementation of the Fluid module system supports only those features needed for the case studies performed to date. This includes the features most relevant to analysis developers, but omits many features that would benefit programmers. It would be particularly desirable to implement the missing pieces to enable experimenting with the complete system. We should also investigate bringing this design to the attention of the JSR294 Expert group on Modules for Java. Two missing features of particular note are integration of modules and layers (nesting each within the other) and automatic transformation to move code from non-leaf modules into newly-created leaf modules.

Usability improvements As part of its ordinary operation, the module analysis discovers the actual cross-module references in the as-written code. During the process of dividing an existing system into modules, these actual cross-module references represent a useful approximation of the "intended" module interfaces. We could improve usability substantially by implementing a visibility annotation wizard that started with the as-written references and assisted the user with the process of adding annotations to "make it so." A second important usability improvement would be the addition of a "module-scoped" file that could be used to hold scoped annotations. This improvement would permit a further order of magnitude improvement in the number of annotations per KSLOC needed to thread color an application (see Section 3.5.2).

7.5.4 User studies

One result of our various case studies is evidence that thread coloring and the prototype assurance tool are practicable and useful. Carefully designed user studies could allow us to measure the actual effort required for thread coloring analysis, and to study developer comprehension of the concepts. Such studies would add valuable data to our current understanding. It would also be valuable to perform a longitudinal study to measure the benefit provided to an organization by thread coloring and the prototype assurance tool.

Nomenclature

API	Applications Programming Interface. Also used as shorthand for the visible interface of a Module.
Applications Programming Interface	The group of visible methods, variables, types, and other entities that make up the supported interface between a library or framework and its clients.
AST	Abstract Syntax Tree. A tree representing the syntax of a program or program element.
BDD	Binary Decision Diagram. A compact and efficient representation for Boolean expressions, in which each variable need be tested no more than once for any expression.
CFG	Control-flow Graph.
CG	Call Graph. A graph that represents the caller-callee relationships between method-like entities in a program or program fragement.
Cohesion	The degree to which data declarations and subroutinees of a compilation unit (CU) are conceptually related, based on information known at the end of high-level design.
color	An uninterpreted tag that represents a particular use or purpose for a thread.
color bindings	Threads may be associated with zero or more colors (or roles) at any time. These associations are formed when the thread reaches a program point annotated with a grant annotation. Intuitively, when a thread is associated with a role it has taken on that role. The color binding of a thread at a particular instant of execution is the set of roles the thread is currently performing.
Color constraint	An expression representing the maximum acceptable color environment permitted to execute a method. The constraint also provides the initial color environment for the method body.
color environment	The set of all possible color bindings between roles and threads at a particular program point or data reference. Represented as a Boolean expression with color names as variables.
Composability	The ability to combine separate analysis results from multiple modules into a single unified result.
Coupling	The extent to which a software part is related to other software parts. A relation between an individual software part and its associated software system.
CU	Compilation Unit
deponent	One who deposes or makes a deposition under oath; one who gives written testimony to be used as evidence in a court of justice or for other purpose. [2]
eAST	Eclipse Abstract Syntax Tree
exported interface	That portion of a module's contents that are visible outside the module. Also known as the visible interface or public interface of a module.
fAST	Fluid Abstract Syntax Tree
home module	The home module of a Java entity is the module that most closely contains it.
IR	Internal Representation. Typically the Fluid Internal Representation in this thesis.

[2]"deponent, *n.*[2]" *The Oxford English Dictionary.* 2nd ed. 1989. Oxford University Press.

JVM Java Virtual Machine

KSLOC Thousand (K) Source Lines of Code

Module A container for source code (or other modules in a hierarchy) with a defined interface, which is visible from outside the module. All contents of the modules not part of the interface are hidden. The purpose of modules is to control static visibility and (secondarily) to provide a medium-sized granularity for static analysis.

public interface That portion of a module's contents that are visible outside the module. Also called the exported interface of a module.

Root module In the Fluid module system, the root module is the empty module at the top of the module hierarchy.

SLOC Source Lines Of Code

Synchronization Acquiring and releasing locks to protect the consistency of shared state (in Java).

Thread usage policy Rules a client program must follow to assure correctness with respect to concurrency.

TMS Truth Maintenance System. A system that provides support for dependencies amongst the data items it contains, usually including transitive invalidation. This ensures that when an item is marked invalid, all dependent items are also invalidated.

top-level module Any module whose immediate ancestor is the root module.

Bibliography

[1] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. Springer-Verlag, 2004. 7.1

[2] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded java programs. *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, pages 68–75, 2001. doi: 10.1109/ASWEC.2001.948499. 7.1

[3] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13)*, pages 217–226, September 2005. 4.6.2

[4] Joseph Bowbeer. The last word in swing threads working with asynchronous models. Sun Developer Network, May 2005. URL `http://java.sun.com/products/jfc/tsc/articles/threads/threads3.html`. 1.3.3, 2.2, 2.3.6

[5] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 56–69, New York, NY, USA, 2001. ACM. ISBN 1-58113-335-9. doi: http://doi.acm.org/10.1145/504282.504287. 1.2, 7.1

[6] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1. doi: http://doi.acm.org/10.1145/582419.582440. 1.2

[7] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Note in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. 1.5.2

[8] S.; Basili V.R. Briand, L.C.; Morasca. Defining and validating measures for object-based high-level design. *Transactions on Software Engineering*, 25(5):722–743, Sep/Oct 1999. ISSN 0098-5589. doi: 10.1109/32.815329. 3.2.1

[9] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8. doi: http://doi.acm.org/10.1145/286936.286947. 1.5.2

[10] John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: A Rational Module System for Java – and its applications. In *OOPSLA'03*, pages 241–254. ACM, ACM Press, 2003. 3.2.2, 3.6.1

[11] DDC-I. Ada product sheet. Web Download, October 2007. URL `http://www.ddci.com/Downloads/product_pdfs/Ada.pdf`. Current as of Oct. 2007. DDCI sells and maintains some of Tartan's Ada compilation systems. 3.4.4, 3.6.1

[12] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/

945445.945468. 1.2, 7.1

[13] George Fairbanks. *Design Fragments*. PhD thesis, Carnegie Mellon, 2007. URL `http://reports-archive.adm.cs.cmu.edu/anon/isri2007/abstracts/07-108.html`. 4.3.2, 10

[14] George Fairbanks, David Garlan, and William Scherlis. Design Fragments make using frameworks easier. volume 41, pages 75–88, New York, NY, USA, 2006. ACM. doi: http://doi.acm.org/10.1145/1167515.1167480. 10

[15] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *PLDI '00*, New York, 2000. ACM Press. 1.2, 7.1

[16] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag. ISBN 3-540-41791-5. 1.4.1, 1.5.2

[17] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02*, New York. ACM Press. 1.2, 1.5.2

[18] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. *SIGPLAN Not.*, 37(5):234–245, 2002. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/543552.512558. 1.2, 6.2

[19] Matthew Flatt. PLT MzScheme: Language Manual. Technical Report TR97-280, Rice University, 1997. 3.6.1

[20] M. Franz. The Programming Language Lagoona: A fresh look at Object-Orientation. *Software - Concepts and Tools*, 18(1):14–26, 1997. 3.6.1

[21] Ben Galbraith. An introduction to SWT. JavaLobby Expert Presentation Series, November 2004. URL `http://www.javalobby.org/eps/swt_intro/`. Slides and voice track. 4.7.2

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2. ISBN:0-201-63361-2. 4.6.1

[23] Aaron Greenhouse. *A programmer-oriented approach to safe concurrency*. PhD thesis, Pittsburgh, PA, USA, 2003. Chair-William L. Scherlis and Chair-Thomas Gross. 1.2, 1.7.2, 7.1, 7.3

[24] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 205–229, London, UK, 1999. Springer-Verlag. ISBN 3-540-66156-5. 1.4, 4, 2.3.14, 2.3.14, 3, 6.4.3

[25] Aaron Greenhouse and The Fluid Team. *An Introduction to Declaring Design Intent in Fluid*. Carnegie Mellon, Institute for Software Research, 5000 Forbes Ave, Pittsburgh PA 15213, July 2006. URL `http://www.fluid.cs.cmu.edu:8080/Fluid/annotation-handout.html`. 17, 2.3.14, 4.5.2

[26] Aaron Greenhouse, T. J. Halloran, and William L. Scherlis. Using eclipse to demonstrate positive static assurance of java program concurrency design intent. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 99–103, New York, NY, USA, 2003. ACM. doi: http://doi.acm.org/10.1145/965660.965681. 1.5.2, 1.7.2, 6.2

[27] Aaron Greenhouse, T. J. Halloran, and William L. Scherlis. Observations on the assured evolution of concurrent Java programs. *Sci. Comput. Program.*, 58(3):384–411, 2005. doi: 10.1016/j.scico.2005.03.002. 1.2, 1.4.1, 4, 7, 2.2, 7.1

[28] D. Grune. Concurrent versions system, a method for independent cooperation. Technical Report IR 113, Vrije Universiteit, Amsterdam, 1986. 3.2.1

[29] Timothy J. Halloran. *Towards a Scalable and Adoptable Approach to Analysis-based Verification of Mechanical Program Properties*. PhD thesis, Carnegie Mellon, Pittsburgh, PA, to appear. 3.2.1, 6.3.2, 6.4.3, 6, 6.5.1, 6.8

[30] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. doi: http://doi.acm.org/10.1145/996841.996844. 1.2, 7.1

[31] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999. URL `citeseer.ist.psu.edu/igarashi99featherweight.html`. 5.1, 5.2.1, 5.2.3, 5.5.3

[32] Adacore Inc. Gnu ada translator (gnat). URL `http://www.gnu.org/software/gnat/gnat.html`. Ada compiler fully integrated with gcc. 3.6.1

[33] A. Loginov J. Choi and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical Report RC22146, IBM Research, 2001. 1.2, 7.1

[34] Java Developer Connection. Bug id 4138730: Provide apis in awt for porting multi-threaded guis to a single-threaded model. `http://developer.java.sun.com/developer/bugParade/index.jshtml`, 1998. 1.3.3

[35] Java Developer Connection. Bug id 4143834: Swing thread implementation causes notifier to lock. `http://developer.java.sun.com/developer/bugParade/index.jshtml`, 1998. 1.3

[36] JSR294-Expert-Group. JSR 294: Improved modularity support in the JavaTM programming language. 3.4.4, 3.6.1

[37] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*. Kluwer, 1999. 1.2

[38] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1127878.1127884. 1.4.1

[39] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq SRC, Palo Alto, CA, 2000. 1.2

[40] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: new-age components for old-fasioned Java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 211–222, New York, NY, USA, 2001. ACM Press. doi: http://doi.acm.org/10.1145/504282.504298. 3.6.1

[41] Hans Muller and Kathy Walrath. Threads and swing. Sun Developer Network, September 2000. URL `http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html`. 1.3.3, 2.2

[42] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41 (6):308–319, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1133255.1134018. 1.2, 7.1

[43] Joseph M. Newcomer. Using worker threads, Jan 2001. URL `http://www.flounder.com/workerthreads.htm`. 1.3, 1, 7.1

[44] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking:. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 11–20, New York, NY, USA, 2002. ACM. ISBN 1-58113-514-9. doi: http://doi.acm.org/10.1145/587051.587054. 1.2, 1.4.1, 1.5.2

[45] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 229–239, New York, NY, USA, 2002. ACM. ISBN 1-58113-562-9. doi: http://doi.acm.org/10.1145/566172.566213. 1.5.2

[46] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):

1053–1058, December 1972. doi: http://doi.acm.org/10.1145/361598.361623. 3.2.1, 1

[47] Tom Perry. Blue panels with swing FIXED. `http://lists.apple.com/mhonarc/java-dev/msg16355.html`. 1.3, 1.3.2, 7.1

[48] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. *SIGPLAN Not.*, 41(6):320–331, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1133255.1134019. 1.2

[49] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 14–24, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. doi: http://doi.acm.org/10.1145/996841.996845. 1.2, 7.1

[50] IBM Rational. Clearcase. URL `http://www.ibm.com/software/awdtools/clearcase/`. 3.2.1

[51] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69. ACM Press, 2001. doi: {http://doi.acm.org/10.1145/378795.378811}. 3.6.1

[52] ISO SC22/WG9. *Ada 2005 Annotated Reference Manual. ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1, 2006.* International Standards Organization. URL `http://www.adaic.com/standards/ada05.html`. 3.2.1, 3.2.2, 3.4.4, 3.6.1

[53] Mary Shaw. Some patterns for software architectures. pages 255–269, 1996. 3.2.1

[54] Kerry Shetline. Skyviewcafe, 2000-2007. URL `http://www.skyviewcafe.com/`. 4.4.1

[55] Saurabh Srivastava, Michael Hicks, and Jeffrey S. Foster. Modular information hiding and type-safe linking for C. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 3–14. ACM Press, 2007. doi: http://doi.acm.org/10.1145/1190315.1190319. 3.6.1

[56] Andreas Sterbenz and Alex Buckley. Strawman proposal for JSR294 superpackages. Accessed March 2008, March 2007. URL `http://blogs.sun.com/andreas/resource/superpackage_strawman.html`. 3.6.1

[57] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the 1993 USENIX Winter Technical Conference*, pages 97–106. USENIX, 1993. 1.2, 7.1

[58] Subversion. Subversion. URL `http://subversion.tigris.org/`. 3.2.1

[59] Dean F. Sutherland, Aaron Greenhouse, and William L. Scherlis. The code of many colors: relating threads to code and shared state. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 77–83, New York, NY, USA, 2002. ACM. ISBN 1-58113-479-7. doi: http://doi.acm.org/10.1145/586094.586109. 4, 1.7.2

[60] The AspectJ Team. The aspectj programming guide. Technical report, Xerox Palo Alto Research Center, 2004. URL `http://eclipse.org/aspectj/doc/released/progguide/index.html`. 4.5.2