# Conflicts of Interest: Approaches to Extensible System Design

Benjamin Lerner

April 1, 2009

**Abstract**

An extensible system permits multiple third-parties to add to, revise, or fundamentally alter the functionality provided by the base system. System extensibility is a spectrum: on one extreme is a completely hard-coded system with no extensions permitted; at the far extreme is a system that provides the barest minimum of hard-coded support and expects all functionality to be provided as extensions. As the capabilities of extensions grow, so too does the likelihood and severity of conflicts among extensions. Working with these systems thus requires a solid understanding of the *extensibility model* they provide: what exactly are extensions capable of doing, both to the underlying system and to each other, and what can go wrong when extensions interact?

Recently, a new extensible system has grown in importance: the web browser. Browser vendors have experimented with various extensibility mechanisms, but little systematic work has been done to understand the design space. In this report we examine several types of systems that prominently deal with extensibility—aspect-oriented programming, operating systems, feature specification, and security monitors—and explore the extensibility model taken by several illustrative papers in these areas. We develop a classification scheme for the extension design space and show that in each area the scope of extensibility can vary widely. Finally, we position the web browser in this design space, and show techniques from these systems may be adapted to the browser.

## 1 Defining extensibility

Intuitively, an *extensible system* is one that permits later revision of the previously-designed base system: additions to, improvements upon, or replacements of existing functionality. Extensibility is everywhere, in varying degrees. On the minimalist extreme, simple music players admit additional codecs to support new encoding formats, but for example no extension may provide internet support. More flexibly, office suites embed scripting languages that can encode fairly powerful computations, but for example cannot add support for new file formats. On the far extreme, some systems are practically nothing but extensions—other than a small runtime core, the emacs editor is a default collection of macro packages, written the same way as the non-default packages known as extensions.

In the past few years, people have come to expect their *web browser* to be an extensible system as well, adding toolbars, social-network customizations, interface tweaks, and many other personalizations to adapt the browser to their needs. Currently, the undisputed leader in customizability is the Firefox browser, which was architected so that extensions could leverage the same technologies for their interfaces and functionality as web pages and Firefox itself use for their own appearances and behaviors. One fascinating consequence of this design choice is that nearly *anyone* can write a Firefox extension—the learning curve is only modestly higher than that for publishing a web page. Further, it is extremely easy for end users to install extensions: `addons.mozilla.org` hosts over six thousand distinct extensions, and over one billion extensions have been served as of 2008. As a result, for the first time we have a software system

that can be extended by very amateur programmers, and where the interactions between extensions cannot feasibly be tested by anyone, let alone independent extension authors. Indeed, in the current forty top-ranked extensions at least five directly conflict with each other.

The paper examines the *extensibility problem* as it pertains to web browsers: understanding how extensions interact with one another and how best to resolve conflicts that arise. We will examine prior work in four other areas of research: aspect-oriented programming and operating systems, which focus primarily on defining extensions; and feature specification and security monitors, which focus mainly on resolving conflicts. To frame the discussion, we introduce a set of criteria against which to describe an extensibility system. Previous efforts have classified system-specific criteria for operating systems [33, 116] and feature specification [74]; our criteria generalize and combine these to apply to all systems examined here. In this paper, extensions will be examined based upon their:

- **Design considerations**: What *behavior* does the extension have—e.g., new policies, better performance, new functionality? Alternatively, what *semantics* are designed into the extension system? Who is the *author*—e.g., the base system's designer, an external developer, or an amateur? When is the extension *integrated* with the base system—e.g., at design, build, install, load, or run time? Finally, how much *cooperation* does the extension need from the base system—e.g., can the base system be written oblivious to future extensions, or must it accommodate them during its design?

- **Extension abilities**: What base system *resource* do extensions target—e.g., the set of supported hardware, or the available security policies? Given that, how *pervasive* is the extension—how much of the system may be modified? How *granular* can extensions be—e.g., can they replace single lines of code, or only entire subsystems at once? In what ways can extensions *compose*? Finally, what *interactions* are possible between multiple extensions or between each other and the base system, and what guarantees can be made when no features interact?

- **Troubleshooting techniques**: What *conflicts* are possible—i.e., which interactions are undesirable? When can conflicts be *detected*—e.g., at design time, via runtime testing, or via user problems? Can individual extensions be be checked *modularly* for conflicts? Finally, how are conflicts *resolved*—e.g., by restricting the action of the extensions, by manually composing them into a corrected composite, or by rewriting the extensions to cooperate?

The rest of the paper is organized as follows. The breadth of each area precludes defining *the* extension model for each domain, so in Section 2 we first give an overview and a strawman extension model of the four systems types mentioned above. We then survey several salient approaches across these areas that focus on design considerations (Section 3), extension abilities (Section 4) and troubleshooting techniques (Section 5). Finally, in Section 6 we approach the browser design space with these reference points as guides. Section 7 concludes.

# 2 Overviews

To ground our discussion of extensibility, we will first give a general overview of aspect-oriented programming, operating systems, feature specification and security monitors. We describe the systems in terms of our classifications above, then define a generic strawman extension model for each system, to contrast with each specific instance presented later.

## 2.1 Aspect-oriented programming

Fundamentally, an *aspect* extends the control flow of an existing program by declaring *what* additional work must be done *when* certain conditions arise. The primitive hook for extension within AOP is the *joinpoint*, which describes an "interesting" moment of control-flow such as a function entry or exit. *Pointcuts*[1] define groups of joinpoints, usually via simple set operations. Specifying *what* action to take at a given pointcut is known as *advice*. Multiple pointcut–advice pairs may be combined to define a single aspect, which is defined compactly and separately from the *mainline* code (i.e., the non-aspect remainder of the program), until being *woven* together at compile time. Advice may apply *before*, *after*, or *around* a pointcut, though these forms may be simplified: as presented in Walker et al. [126], these can all be compiled to idioms of joinpoints, where advice is executed *at* the moment the joinpoint is reached.

*An idealized AOP system permits extensions to modify the state and control flow of the mainline program via aspects that advise function calls. Advice is authored by the same developers as the mainline system, and is integrated at build time. The mainline code need not cooperate to enable advice, though care must be taken for the advice to work properly: conflicts may arise when multiple advice apply to a particular pointcut (detectable by the compiler), when aspects overlap in function, or when changes in the mainline break an aspect (detectable only by testing); all require programmer intervention. When all aspects are compatible with each other and the mainline, the semantics of the woven program is predictable and independent of the weaving order (modulo any explicitly specified ordering constraints.)*

The above definitions are very much simplified; each part of these definitions can be varied, yielding systems with greater or lesser flexibility—and impact on ease of analysis for conflict. The set of available joinpoint kinds (granularity) differs considerably between AOP implementations; while AspectJ contains a predefined set [75], they could be programmer-specified labels interspersed anywhere within the code [126]. Moreover, while AspectJ's joinpoints are focused on OO-style code (e.g. a special joinpoint for object constructors), there is nothing inherently object-oriented about aspects, and indeed aspects can be formalized for functional languages as well [28, 29, 36, 126]. Similarly, the flexibility of pointcut specification differs between implementations (e.g., [19, 36, 40, 75, 126]), and is a key factor in the utility of the AOP approach. More complex systems permit pointcuts that can examine the stack[2]; others permit state machine-like inspection of the history of the computation. AOP languages may choose whether aspects are second-order,

[1] Confusingly, Douence et al. [36] use the term *crosscut*; Rinard et al. [107] uses the latter as an adjective, so we will avoid using it in this report.

[2] Such pointcuts include AspectJ's `cflow` construction, which matches when the specified procedure is present on the stack; arbitrary pattern matching against the stack is possible [27].

syntactic constructs baked into the language [75], are extensible but still second-order language constructs [19], or are first-order values definable and manipulable within the language itself [40]. Advice can be dynamically installed for a given scope, or lexically inserted at compile time. One point of common disagreement appears to be the appropriate weaving order for aspects. Aldrich [5] applies advice in a LIFO order, while Dantas and Walker [27] take the exact opposite approach. AspectJ [75] defines a complex (and incomplete) set of rules governing weaving. Other systems (e.g., Douence et al. [38]) permit programmer-specified weaving orders. The only commonalities seem to be the need for clearly specifying the semantics of the system under study, and to choose the appropriate heuristic for the task at hand. Finally, the pervasiveness of the advice itself can be varied, by varying the environment available to the advice.

## 2.2 Operating systems

Operating systems are the lowest-level software interface between physical resources and software. They are traditionally responsible for multiplexing those limited resources between multiple clients, and for abstracting the details of those resources into a convenient programming interface. In one sense all operating systems are trivially extensible: they permit an unbounded set of user-level programs by exposing an abstracted view of the machine. In the following discussion, we are uninterested in these applications, and focus rather on more fundamental changes in functionality, which require operating beneath the user-visible abstraction layers. The case for kernel extensibility has been made numerous times (see for instance [7, 42, 44, 60, 81, 82, 111]) and can be summarized by the simple observation that no closed system can be all things to all people. An extensible OS therefore must expose enough control over its internal structures to permit tailoring their behavior to applications' needs. Such extensibility must be tempered by concerns for performance and safety—if the extensibility mechanisms are too expensive, functionality would be better written in user-mode on a system with a cheaper mechanism; if unsafe, the system behavior becomes unmanageable.

*An idealized extensible OS permits kernel extensions to expand the resources (e.g., new hardware or better performance) available to the rest of the system, via modules that expose new APIs. Extensions are typically authored by the vendors of those new resources, rather than the OS developer, and are loaded dynamically when the OS boots. The OS itself must expose some hooks so modules can rely upon a standard interface. Modules may interact through any shared state of the OS, and may conflict if multiple modules expect exclusive control over some resource; such conflicts are detectable only through detailed testing, and require developer intervention to fix. Compatible extensions may be loaded in any order (unless otherwise specified), and will not destabilize the system.*

This focus on extensibility leads to the microkernel approach to OS architecture: as explained by Liedtke [86, 87], microkernels should include in the kernel only that which is necessary to implement the remainder of the system; everything else need not be in the kernel. (Note that this is distinct from whether extensions may *run* in the kernel memory space or not, a choice which varies among different microkernels—for instance, device drivers are not

considered part of a microkernel, but may run with kernel privileges.) Since operating system efficiency is paramount, endless clever mechanisms have been proposed to provide extensibility.

Broadly, all OS extensions, regardless of mechanism, supply a new method of interacting with some resource of the system: they extend the set of resource access protocols. The space of extensions therefore depends heavily on the granularity of the resources exposed by the existing kernel. If a kernel exposes a file system abstraction, for instance, then extending it might entail permitting new access-control mechanisms or policies for files. If the kernel only exposes a disk-level abstraction, extensions might include file system implementations themselves.

## 2.3 Feature specification

The larger a system grows, the greater the chance that seemingly unrelated parts interact in unexpected ways. At the concrete level of code, interactions may make the codebase difficult to improve over time; at a higher level, interactions may threaten the correctness of the system's user-visible behavior. *Feature specification* aims to address this higher-level concern: a *feature* is an informal, self-contained, user-visible piece of functionality, whose behavior may be rigorously *specified* using a number of techniques. These specifications may be checked against each other and against axioms of the base system, to detect unwanted interactions—conflicts. This *feature interaction problem* has been intensely studied by the telecommunications industry: as they roll out new features (e.g., call waiting, caller ID) they must ensure that previously-working features do not break unexpectedly.

Unlike AOP or operating system extension mechanisms, feature specification is is a step removed from the extensible systems it is used to model. While the underlying system therefore has to contend with granularity, pervasiveness, integration and other architectural issues, feature specification simply uses a very expressive (and therefore granular) logic to express whatever properties the underlying system ought to have. Likewise, the integration of multiple feature specifications mimics whichever integration technique the underlying system uses. Feature specification concerns itself almost exclusively with the troubleshooting part of extensibility.

*An idealized feature specification effort is a collection of abstracted, checkable formal models that represent the desired behavior of the implemented system. Features can specify fine-grained responses to individual situations, and may require behavior that spans the entire underlying system. New features can be added to the collection over time, and the collection can be checked anew to ensure different features do not break each other's specifications. Interactions occur when feature specifications require distinct reactions to the same situation; conflicts occur when interactions were not expected. Conflicts are resolved by prioritizing features (i.e., restricting their scope) or rewriting them; these resolutions are then reflected in the eventual implemented program. When all features are compatible, all behaviors demanded of the system hold—a guarantee only as strong as the specification effort.*

As should be no surprise, there are a wealth of techniques for defining and modeling features, and defining, detecting

and resolving conflicts. Keck and Kuehn [74] and Calder et al. [22] give several axes to examine the literature; we will focus on their *causal view*, where the goal is to identify a witness to the cause of the interaction, rather than classifying interactions by when during development they might have been introduced or managed. Additionally, we will focus solely on design-time specification techniques. Though there are several efforts at run-time feature specification (e.g. [15, 16, 21]), these feature-managers look very similar to runtime security monitors discussed later.

The essential idea behind feature specification is to describe the behavior over time of the features in the system: a temporal property is true *now* if at some time *later* a sub-property holds. All temporal logics can express several powerful notions relating to time: e.g., a property $p$ holds *always*, *eventually*, *until* a property $q$ holds, or *next* time. Deterministic automata encode (nearly) the same notions via sequences of states. (Some temporal logics can also express properties concerning nondeterminism; we ignore the details here.) For both mechanisms, all feature specification work focuses on predicates that will be satisfied *infinitely often*—temporally finite properties are a special case as they trivially can be converted to infinite ones. Depending on the problem formulation, conflicts arise either when two specifications are simultaneously enabled infinitely often (where each feature thinks it has sole control but doesn't), or when combining two specifications admits no solutions at all (no system could satisfy both).

## 2.4 Security monitors

Security monitors are necessary whenever a user (or a runtime system) does not trust another piece of code to run within some prescribed bounds. Monitors supervise the execution of untrusted code and intervene when necessary to maintain the desired *policy* of the system, accepting all program executions the policy deems good and rejecting the rest. Most practical policies are *properties*: they judge a program execution in isolation and not relative to other executions. There are two fundamental kinds of properties: *safety* properties that ensure "nothing bad ever happens," and *liveness* properties that ensure "something good eventually happens." Thus, never accessing `/etc/passwd` is a safety property, while always closing all open files is a liveness property. Obviously, these two properties can be conjoined to yield a non-safety, non-liveness property; surprisingly, *all* reasonable[3] properties can be written in this form. [91] All security monitors can thus be classified by whether they support safety, liveness, or a combination of both property types.

*An idealized security monitor enforces a single policy over a given program, by observing and mediating any security-relevant actions taken by the program. The monitored program is oblivious to the presence of monitors; in fact this obliviousness ensures that the monitor is not subvertible by the program. Policies are usually written by users or system administrators, rather than the program's developers, and their granularity depends on the monitoring technique used. Multiple policies must be combined before the monitor can enforce the composite; the default is merely to intersect the policies and reject the program if any monitor rejects it, which means conflicts among policies may*

[3]Formally, a reasonable property is a decidable predicate over program executions that at minimum is true for the empty execution.

*cause fewer programs than expected to be valid. Conflicts among policies are simply mismanaged user expectations; the monitor will run properly regardless of what cumulative policy it enforces.*

The distinctions in security monitor implementations lie mainly along the integration axis: static access controls for specific policies can be enforced at build time (of either the mainline program or the policies) [11, 41, 68, 69, 115]; execution monitors can supervise the program's runtime; inlined reference monitors activate (at the latest) at load time. Among the dynamic approaches, few systems actually consider their extension model in any detail (e.g., [3, 31, 32]): how might multiple policies compose for a given target? For those that do, designs must consider whether policies apply only to the mainline program, or whether they layer over each other. (Consider combining a policy limiting memory usage with one limiting processor time: if for a tiny, fast program the CPU-usage policy took too much memory to enforce, should the memory-usage policy abort the program due to the CPU policy, or accept the well-behaved mainline program?) Systems that ignore policy composition implicitly assume that policies inspect only the original program.

As was mentioned earlier, security monitors are one of AOP's strengths, describing a whole-program property that should be expressible in a concise and easily-understood way. Unsurprisingly, AOP is one of several implementation techniques for security monitors [31, 59, 117], though binary rewriting has been explored more thoroughly [12, 25, 46–50]. Additionally, we have seen security monitors appear as access control mechanisms in operating systems (e.g., [58, 93, 108]). These systems usually enforce only one policy at a time, obviating the need for policy composition.

# 3 Design considerations

Of the design considerations described above, authorship and integration time tend not to vary widely within each systems type. However, there is extensive work on formalizing semantics for these areas; we mention several efforts here.

**Aspects: Language design** Common wisdom [54] suggests that aspects are most effective when the mainline code is *oblivious* to the effects of the aspects, i.e., written without consideration of or accommodation for future aspect extension, though this is somewhat oversimplified [63], and demands understanding what effects aspects can have on a program. There is extensive work on formalizing the foundations of aspects [4, 26, 28–30, 36, 40, 63, 64, 70, 90, 110, 113, 123, 126–128]. Several of these try to apply aspects to a typed (e.g. [28, 30, 90]) and/or higher-order functional [40, 94] setting. The former, while technically demanding, ensures that well-typed aspects cannot cause well-typed programs to go wrong. The latter adds enormous flexibility by permitting the definition of new forms of pointcuts as needed, though it makes compiling and optimizing the woven code much harder. There is also work that examines the semantics of extending one language with multiple aspect mechanisms themselves. [77] This level of extensibility does not bear on the browser extensibility model, and will not be addressed further here.

**Feature specification: Logic choice** As with aspects, nearly every feature specification effort proposes a slightly different mechanism (among others, [18, 39, 62, 95, 103, 114]), and the subtleties distinguishing one temporal logic from another are not important here (indeed, no real consensus has been reached; see for instance [1, 2, 43, 80, 124]). Choosing a particular logic ultimately depends on knowing what conflicts need to be expressed for a given problem.

**Security monitors: Complexity** Before addressing the pragmatic concerns of how security policies might be specified, it is necessary to know what policies are expressible and enforceable. One line of research [6, 88, 89, 112, 125] precisely defined the complexity class of properties enforceable by security automata: if the only remedy available to a monitor is to terminate the offending program, then such a monitor can enforce precisely the safety policies; if a monitor can also forge or suppress behavior of the offending program, then a wider class of transaction-like policies are available. Understanding this class, and under what operations it remains closed, informs how policies may compose.

# 4 Extension abilities

Though every project tweaks several of our organizing criteria, our discussion below describes efforts most notable for varying some facet of extension ability. The precise extended resource and set of interactions depends on the particular approach, so we organize our discussion around the other extension ability axes: defining safe idioms for their usage (indirectly varying their pervasiveness), exploring composition mechanisms, and refining extension granularity.

## 4.1 Safe idioms

**Safe AOP idioms** There are several efforts to define safer idioms of programming with aspects [5, 10, 27, 71, 100, 107]. This line of work seeks to place limits on either pointcuts [5] or advice [27] to restrict their unprincipled tampering with the mainline program. These two approaches roughly correspond, in other systems settings, to narrowing the client API and restricting permissions on client actions.

Aldrich [5] takes as his starting point the fragility of pointcuts: by hooking into the control flow of a program, they necessarily rely on that structure remaining unchanged across versions. Moreover, such deep hooks prevent most local reasoning about abstraction boundaries. To address this, Aldrich restricts his set of primitive joinpoints to include only call sites of *declarations*, rather than of all functions, a distinction that exposes only named, exported functions to advice, while anonymous lambda expressions and functions hidden behind module boundaries are immune. Within a module's definition, all call sites are available as joinpoints; "open modules" can choose to expose some of these otherwise-hidden joinpoints as part of their signature[4], making them targetable by external advice. External calls to

[4]The use of modules, opaque signatures, and functors derives heavily from ML's module system, but includes width, depth and transitivity subtyping rules among signatures, not described here.

functions in the signature are advisable; intra-modular calls to those functions are not, unless they are exposed through an explicit addition to the module signature. Doing this makes the exposed joinpoints "a part of the API," implying they will be stable in the face of future internal changes. Note that this comes at a price: the module author is no longer oblivious to the potential for future advice.

This stability is formalized by Aldrich's treatment of the *equivalence of modules*, which for our purposes is the key contribution of this paper. A revision of a module is equivalent to the original when it does not break aspects that previously worked. More formally, "equivalent functions must not only produce equivalent results given equivalent arguments, they must *also* trigger advice on client-accessible labels in the same sequence with the same arguments"—revisions must preserve the behavior of their declared joinpoints, and the social conventions regarding API change come into effect. The essence of the open module approach has been adopted, for example, by the Eclipse platform [17], where plugin authors explicitly declare extension points for future plugins to use. Additionally, extension points must be deprecated before they are removed or changed. By contrast, in current browser designs, extensions frequently break between minor version changes of the browser; while effort is made to prevent needless problems, such brittleness is endemic when extension points are not promised to be stable.

Dantas and Walker [27] take the dual approach, placing restrictions on what advice can do with a source program. They focus on when is it possible to be certain that aspects do not *interfere* with the mainline program. So-called harmless aspects may observe the execution of a program and may influence its termination behavior (e.g., they can terminate it in response to an error condition) but they cannot influence its computed results; this definition preserves partial-correctness properties of the mainline program, in the face of changes to the aspects woven into it. To achieve this noninterference result, the authors define an information flow-like type system that ascribes protection domains to code. Their goal is an integrity property, so intuitively they ascribe a high-protection domain to the mainline code and low-protection domains to the aspects; the typing rules then guarantee that non-unit (i.e., information-carrying) values cannot flow back from the advice to the mainline code.

The key contribution here is their simplification of information-flow to the aspect setting. Harmless aspects are *consumers* of the state produced by the mainline program, and the type system enforces the producer/consumer split. But while their exposition in the paper only concerns separating aspects from mainline code via low- and high-integrity levels, their framework supports a standard lattice. Such a general stratification could support the layering of aspects, guaranteeing that for a weaving order compatible with the lattice order, aspects can be protected from each other as well. The primary drawback to this particular approach is its rejection of any other kind of aspect. Other work [24, 107] developed a classification of aspects supporting both consumers and the dual producers, as well as independent and interfering varieties. In the web browser setting, extension authors typically write extensions that interfere with the mainline browser, though potentially most extensions may not write to or interfere with each other.

**Static OS extensions: Aspects and code management** There is a small line of research porting the code-management facilities of AOP to OS development. In a short position paper, Fiuczynski et al. [55] observe that while several research OS projects are obviously extensions to some mainline OS, they are equally well *aspects*—each one a single concern cutting across the codebase to implement a feature. Indeed, matching our taxonomy, these extensions are build-time integrated, pervasive and fine-grained, and provide new functionality and new policies. Like all aspects, they extend the control flow and state of the mainline kernel. However, the authors lament that the current notation of these extensions as patch sets—collections of the syntactic differences between the mainline and the extension—is inadequate. These extensions are semantic units, and therefore ought to be expressed explicitly in an AOP language. Further, patches are notoriously brittle to tiny perturbations in the mainline code, and a more semantics-driven weaving mechanism (explored also in [99]) would ease extension management considerably. (In this spirit, Lohmann et al. [92] built an aspect-oriented embedded OS, and described their successes implementing various memory protection schemes as aspects—unsurprisingly, the AOP approach permitted more flexibility than patch-sets would have.)

In follow-up work, Reynolds et al. [106] examine to what extent the Linux kernel can be "unwoven" into separate aspects, and how amenable it is to extension by advice. They note that *coding conventions* weakly approximate aspects, as preprocessor directives are used to separate code pertaining to only one use-case, and optimistically suggest that an AOP rephrasing of the code is feasible. As an added benefit, by making these aspects explicit, the analytical tools of AOP (some of which we described earlier) can be applied to kernel maintenance issues, including easing the brittleness of applying extensions and identifying when two extensions might cause conflicting changes to the mainline code.

## 4.2 Experiments in composition

**Naccio: Safety properties with minimal composability** Evans and Twyman [50] focus on code safety properties, which ensure that no "bad thing" can occur in a program execution. They first compile safety properties into a policy description, then use program transformation to weave that policy into a target program: they produce a library that wraps the security-relevant system calls, and rewrite all existing calls to instead call the library. Safety policies are straightforward to combine, since they are closed under conjunction. If two policies each prevent something bad from occurring, the conjunction of those policies naturally would prevent both bad things from occurring. If all policies are safety properties, then there can be no conflicts. As described, Naccio's policies are somewhat limited to use, since reasonable policies may wish to prohibit bad things *except* in special circumstances; Naccio permits these "weakened" security policies as well [51]. These weakened policies provide more flexibility without creating new conflict types.

**Polymer: Composable, non-safety properties** Bauer et al. [12] provide a much richer environment for defining security policies. Most notably, they use a more expressive mechanism for defining policies, and they give the policy author much greater control over policy composition. We address each in turn.

Their formalism is based upon edit automata [88, 89, 91], which can delay, suppress or forge security-relevant actions taken by the monitored program. As noted earlier, this permits a much broader class of enforceable "transaction-like" policies. In essence, the monitor can delay the beginnings of a transaction (e.g., logging in to an ATM and asking for money) until a crucial commit action occurs (recording the withdrawal), at which point the monitor can replay the beginning of the transaction and insert the remaining events needed to complete it (logging out), in one atomic action. In this way, the observed behavior of the program jumps from one valid state to another, without the possibility of crashing in a visibly invalid state. Such *renewal* properties strictly subsume the safety properties explored in Naccio.

As noted by the authors [91, section 5], it is not always reasonable to assume the monitor can delay, suppress or forge arbitrary security-related actions. For instance, if a policy requires that programs close all opened files, it may not be possible to delay the open event until the matching close occurs: any intervening I/O operations would block or fail without a valid filehandle. Dually, the monitor cannot forge events that require secrets or time-sensitive responses. These and other obstacles make the edit automata model fairly subtle to implement properly.

However, edit automata enjoy clean composition semantics, which are reflected in the design of Polymer. Instead of the curt actions taken by an automaton (insert, replace, remove, halt), Polymer lets policies make suggestions, and gives authors several combinators with which to merge suggestions: two policies may be intersected, one may take precedence over the other, one may modify the other's suggestion, etc. Thus, Naccio's weakened security policies, which had to be hard-coded into the system, are instead available in Polymer naturally as one policy overrules the other's halt suggestion. Put another way, Polymer supports *higher-order policies*, a flexibility few other systems match.

This flexibility comes at a price: in our model, we note that multiple policies cannot be written modularly in their system without someone eventually manually composing them. Additionally, their language does not expose any conflict checking (i.e., if two policies both address the same situation)—though it is possible that the underlying edit automata may admit an intersection test answering exactly this question. With careful engineering, this may work well in the browser space, as users may reasonably want different means of composing the same sets of extensions; Polymer's approach almost requires the user's input in this case.

## 4.3 Experiments in granularity

**The Exokernel approach: Composable, pervasive but coarse** In some sense, any time a computer spends "running the operating system" is time better spent running user programs, and therefore the design of the operating system should optimize for unobtrusiveness and speed. Moreover, since no operating system can suffice for all users' needs, it would be sheer hubris to select some subset of needs to be addressed and declare the result sufficient. Instead, the appropriate design should therefore be to address *no* needs beyond multiplexing the hardware resources of the system to all users. This rather extreme rationale motivates the Exokernel system.

Exokernel defines an operating system as "any piece of software that the application cannot either change or avoid." It views a kernel as a meticulous switchboard operator, whose sole function is to safely multiplex physical resources [44, 45]. Everything else is policy—and therefore relegated to an extension: how to use the compute, storage or communications resources is left to a "library OS", on a per-application basis, that defines the semantics connecting the low-level hardware and the application needs. Thus a databaseOS could implement a completely customized file system, while a webOS could implement a highly-tuned network interface [56]. If necessary, a common libOS could be implemented to provide emulation of a more traditional operating system [72]. The extension model here is application-centric and very pervasive, but not very granular: a user-mode extension can do anything it wants, using a nearly-native interface to the hardware of the system, but must reimplement an entire libOS to do so. The only conflicts relevant to an exokernel are when a libOS or application hoards resources the system as a whole requires for some other purpose. To that end, an exokernel defines a *secure binding*, a combination of a physical resource name (when possible) and capabilities granting privileges over that resource to the requesting process. These capabilities are enforced by hardware as often as possible, for sheer speed, and in software when unavoidable.

At first impression, the exokernel approach seems a poor fit for browser extensibility. Representing an extreme design point, an exokernel aims to let applications manage and define the semantics of their required resources: speed is a primary concern. But the resources available in a browser, such as incoming web content, profile data, or the user interface, are qualitatively different than those in an OS: these are structured, semantics-laden resources, and are the output of substantial processing effort. Skewing the system design to allow rapid access to slow resources does not address the performance bottlenecks of the system. Speed is not the primary concern, and browser extensions therefore need to be examined in a much more full-featured base environment.

On the other hand, if used correctly exokernels can provide robustness guarantees that current browsers cannot match. If the minimal kernel contains just enough power to draw the window containing content, and provided a narrow interface to talk to a renderer process, the result is fairly close to Chrome's architecture [104]. Alternatively, if the kernel provided most of the rendering logic, but provided a narrow interface for extensions to masquerade as HTTP content providers, the result is the Xax approach [35]. Pushing Chrome's approach to the extreme permits arbitrary content-renderers to be fitted into the browser's UI: there would be no fundamental distinction between rendering HTML and rendering Flash, for instance. These approach does make the browser dramatically more flexible, allowing increased experimentation with new scripting languages or web standards. But at this level a browser ceases being a "browser" and is simply an extensible user program that can display networked information. Additionally, while these extensions expand the set of *content viewable within the browser*, they do not extend the *browser itself*.

**The SPIN approach: fine-grained and wide** On one level, the SPIN project can be seen as an experiment in mechanisms, showing that by requiring extensions to be written in a certain way, one could build a system with easier-

to-use base abstractions, comparable performance, and stronger security guarantees than an exokernel approach [13]. On another level, SPIN is very similar to exokernel, "one step up": providing as part of its ABI some abstractions above the hardware level, but still permitting extensive customization. Philosophically, SPIN adopts the more traditional model of actual kernel-mode extensions, which yields a non-degenerate kernel extension model.

The fundamental choice in SPIN, from which the rest of its architecture falls out as a consequence, is the choice of Modula-3 as the only supported language for writing kernel extensions. Modula-3 is an object-oriented language that has support for modules with opaque types: in a type-safe language, values of an opaque type may be used as capabilities, and SPIN exploits this for its security mechanism. Unlike prior capability systems that required complex data structures and capability checking (e.g., Hydra [129]), SPIN represents capabilities as bare pointers and capability checking degenerates to pointer dereference. A pointer of type `Console.T`, for instance, permits the holder of that pointer to do nothing beyond calling methods expecting a `Console.T`: this is a special case of a more general *parametricity theorem* that in essence states that type-safe code cannot violate the encapsulation of opaque types. SPIN therefore discharges most safety checks *at compile or link time*, rather than requiring runtime hardware or software support[5].

SPIN defines higher-level abstractions than exokernel does: for instance, it defines the rudiments of a thread library and provides a default global scheduler, though it does not impose a specific threading discipline to applications[6]. It also chooses not to expose physical names for memory to extensions, but rather capabilities and virtual addresses; extensions may implement memory management policies manipulating these abstractions (for instance, [109]). To surface these interfaces to extensions, SPIN defines its extension model in terms of *events* and *event handlers*. Event handlers are merely procedures of a given interface; raising an event is essentially just a procedure call. SPIN permits multiple extensions to register handlers for an event[7]. The ordering of such handlers is explicitly undefined, so extensions cannot make assumptions about the presence of other extensions or of their relative ordering. For even finer granularity than handling every event of a given type, extensions may provide guards that pass along only those events they deem relevant. In short, the SPIN model is less pervasive than that of Exokernel but of much finer granularity.

SPIN defines no explicit notion of conflict, though operationally, conflicts among extensions are simply those problems caught by the compiler and linker. But other problems are left unspecified: extensions may remove other extensions' handlers for a given event, and an event's dispatcher returns the value of the last-run handler, which is ill-specified (as handlers are unordered). However, if extensions are well-behaved[8], SPIN's semantics are well-defined: faulty extensions used by one application will not cause another application (or its extensions) to malfunction.

The SPIN model seems a closer fit to the web browser space than Exokernel's does. The ability to define customized protection domains is useful, especially in an environment where we do not yet know the best delineations of those

[5] One runtime precaution is needed: since user code is type-unsafe, raw pointers are wrapped as *externalized references* before escaping the kernel.
[6] At least in user-mode; when an application thread makes a system call, the kernel takes control of its scheduling until the call completes.
[7] In general, with multiple handlers, event dispatch bears an overhead linear in the size of handlers and guards.
[8] Which is rather vague; extensions that are application-specific or disjoint suffice.

boundaries [104, 105]. Additionally, since web browsers currently distribute extensions in source-code form, using a language mechanism to reduce execution overhead is a timely and practical approach. These language-level benefits shine through even more clearly when we consider the Singularity architecture.

**The Singularity approach: fine-grained, not too wide or narrow** Singularity is an ongoing research effort in a clean-slate redesign of an operating system [14, 34, 65–67, 118]. It is predicated on three primary techniques: the exclusive use of type-safe languages, a first-class notion of an application, and a carefully managed channel abstraction for interprocess communication. Together, these techniques permit a robust extensibility model: both user-mode extensions (applications) and kernel-mode extensions (drivers) play by the same rules [67].

An extension in Singularity is defined by a *manifest*, declaring what resources it must be granted, what resources it would like to use, and what channels it exports for interaction. (In fact, the only distinctions between applications and drivers are merely the claim of belonging to the ⟨**driverCategory**/⟩ of extensions, and the dependency on raw hardware resources.) Code is annotated with pre- and post-conditions using Spec# [9], thereby specifying requirements down to the individual procedure level. Channels are essentially two-way pipes that are annotated with state-based contract types, which encode the expected message protocol for the channel. These declarations and types are more than documentation: they are checked by the kernel repeatedly, at compile time, at install time, and at load time[9]. These repeated checks ensure that drivers that cannot run properly do not compile, that drivers that cannot ever successfully load on a particular machine are never installed, and that drivers that cannot run in the current machine state are never loaded. Singularity thereby provides feedback early and often, explaining the cause of an extension failure before it causes a crash. Essentially, the manifest becomes a model for the extension, making it a "self-describing artifact" [118].

For all that it formalizes, Singularity does not explicitly define the notions of extension compatibility or conflict. From the system invariants, compatibility among extensions amounts to requiring only resources already available from the system and existing extensions, and using those resources in type-safe ways. All extensions in Singularity are *sealed*, prohibiting runtime code generation or injection of data into a running process. Therefore *all* communication between processes takes place via channels. Forcing this communication pattern ensures that the temporal guarantees from the types apply in all circumstances. Thus type-safety here is a stronger claim than it is in SPIN. Note that while this does not prevent logic bugs from making extensions handle inputs incorrectly, it does ensure extensions never have to handle incorrect inputs. Dually, we can infer the kinds of conflicts avoided by the system: requiring unavailable resources, sending incorrect inputs to another extension, or breaking abstraction and directly modifying another extension's data.

Relative to the other operating systems discussed above, Singularity permits fewer lowest-level extensions than Exokernel or SPIN allow. For example, Singularity provides thread-manipulation and exchange-heap functions as part of the kernel ABI [66]; Exokernel permits a libOS to define its own threading model [8], while SPIN permits

[9]These declarations imply a need for some form of side-by-side versioning, to ensure that older drivers still have the expected versions of dependencies and prevent "DLL Hell". This is merely an engineering problem; the .NET framework addresses this issue.

extensible memory management [13]. This is mostly because the projects goals are orthogonal—Singularity aims first to improve the dependability and security of operating systems, and this imposes certain design choices that require a more extensive kernel. This decision also simplifies the Sing# type system to ignore such low-level details.

As alluded to when discussing SPIN, language-level techniques seem particularly apropos for the browser space. The choice of manifest-based extensions is particularly appropriate: Firefox's extensions already use a manifest to describe some (minimal) information about themselves, and enhancing these to declare needed resources, à la Singularity, is a natural step. Further, Singularity's type system encodes communication patterns of the extension; currently such patterns in Firefox are merely unenforced documentation. Additionally, Singularity demonstrates a decent compromise between lowest-level extensibility and a strong, expressive type system: by analogy, a browser has no need for replaceable HTML rendering engines. The largest remaining challenge is the lack of a convenient but type-safe API for accessing web pages' contents: similarly to Singularity's drivers and applications, extensions and web pages are defined almost identically, differing only in what resources they are initially allowed to access.

# 5 Troubleshooting techniques

**Conflict detection among aspects** Another broad trend in AOP research focuses on giving developers tools to detect conflicts among aspects or to manually manage their composition [10, 37, 38, 73, 78, 79, 107]. Aspects may conflict with each other in two ways: directly, by overlaying the same joinpoint, or indirectly, by mutating shared state in ways the other extension does not expect. (In AspectJ, advice is arbitrary Java code that can be granted read/write access to the pointcut's parameters (e.g., function arguments), potentially even to private members of classes. Aspects can change the dispatching behavior of a program by adding new interface implementations to classes or even changing the class hierarchy itself [61, 83, 120, 121].) Of the two, indirect conflict will persist independently of the weaving order (two aspects may be disjoint in their pointcuts, yet still overlap in their effects on the program state), and is therefore a coding flaw rather than an aspect-related problem. Therefore, research has focused on capturing weaving-related conflicts.

Douence et al. [37, 38] have developed a framework for modeling aspects whose pointcuts change over the course of the program, and which can carry runtime-derived information through their evolution. They model aspects via finite-state machines, where transitions between states correspond to triggering of pointcuts and associated advice. A global program monitor is responsible for weaving the advice into the mainline and updating the aspects' state as the program runs. The expressiveness of their pointcut language is carefully engineered to keep certain intersection problems decidable; as one consequence, their system cannot express AspectJ's `cflow` pointcut for recursive functions.

Practically all the main ideas of these papers are applicable to the web setting; we focus here on just two. First, they permit aspects to "change interest" and focus on different pointcuts at different times, in a history-dependent way. This is far more general than AspectJ's approach, or even the stack-based inspection examined before, and it remains

orthogonal to the safe-idiom techniques described above. Moreover, in the web setting, the behavior of extensions depends heavily on user interaction, so this facility is a natural fit. Second, the authors define notions of strong and weak independence: whether two aspects are always compatible with each other, or compatible within the confines of a specific mainline program. (They also note that neither independence notion is necessary for aspects to be compatible; commutativity is sufficient as well.) Further, they go on to describe an intermediate independence relation, where aspects may declare a minimal set of requirements on the base program; the aspects will be compatible with any mainline program satisfying those requirements. We will see later that this directly applies to browser extensions.

**Termination conditions** Felty and Namjoshi [53] adopt the use of temporal logic for specifying both the system and the features. (Since temporal logic will eventually be necessary, they view the use of additional mechanisms, such as state machines, as needless additional sources of error.) Specifications in their system are divided into system axioms and feature properties. Their system is not modular: adding new features may require adding to or changing the set of system axioms. (They give the example of adding a new type of call resolution; existing axioms must be enriched to reason about this new status.) Feature properties as presented here are engineered to be intuitive to specify.

Recall that a specification details the behavior over time of a feature: once a feature has reached a given state (a precondition), it will exhibit a certain property until no longer applicable (a postcondition). Felty and Namjoshi give an intuitive internal structure to pre- and post-conditions. Preconditions are modeled as a sequence of events, between which a sequence of properties hold. Postconditions specify what behavior *persists* as a consequence, either *until* some normal resolution or until an exception *discharges* the situation. For example, an informal specification of patients visiting a doctor might say "the patient must *arrive* at the office, and *stay there* until the receptionist *greets* him; once greeted, the doctor *sees* the patient until *finished* or another *emergency* discharges the patient early." This rule schema is simple but redundant: in their case study, the authors never used multiple events in their preconditions, and instead encoded the ongoing properties as persistent postconditions of multiple rules. Additionally, distinguishing the release and the discharge conditions adds no expressive power to the rules, but does simplify their definition of conflict.

A conflict occurs when in every execution where two features are enabled simultaneously infinitely often, the system axioms hold, and the features are not discharged, somehow a feature property does not hold. When conflicts occur, the authors either strengthen the precondition of one feature (making it apply *less* often) or weaken the discharge postcondition (making it apply *more* often), so as to prioritize the other feature. This is again not modular, as adding a feature requires editing the properties of the others to resolve conflicts. For browser extensions, the lack of modularity is problematic. However, this clean skeleton for specifying a weakening of one rule with respect to another will be very useful for resolving feature conflicts and, using the insights we explore below, may be achievable in a modular way.

**Modular checking** Li et al. [84, 85] propose a modular approach to feature checking. They work with *open features*,

which depend upon variables or predicates not under the control of the feature. Their key approach (shared by [20, 23]) is to interpret temporal logic formulas using a *three-valued logic*: variables may be *unknown*, indicating the lack of knowledge that one feature has about another's behavior. When validating feature composition, modularly guess the values for these unknown predicates. Pessimistically, if everything unknown is assumed false, and the feature still validates, then the feature will always validate successfully. Conversely, if everything unknown is assumed true and the feature fails to validate, then the feature will never validate and there is an intrinsic conflict. Otherwise, manual intervention is needed to resolve possible conflicts, by modifying the specifications or by manually composing features.

To achieve *modular* checking, i.e., validation of a single open feature without knowing the precise set of other features being composed into the system, the authors require that specifications not be written obliviously: if a given property $p$ plausibly may be reduced (resp. expanded) in scope by later features, it must be specified proactively as $p \wedge c_p$ (resp. $p \vee c_p$), where $c_p$ is a symbol of unknown value. Moreover, they split the validation process into two: first, they *verify* that the feature's specifications hold in the current product. The same technique as above broadly applies: optimistically set all $c_p$ that reduce scope, and pessimistically set the others, and see if any verifications fail[10]. Second, they ensure that any changes the new feature makes to the product *preserve* existing properties required by prior features. This is their modularity result: once composed and verified, it becomes *later* features' responsibilities to ensure they do not invalidate prior features. In short, while their modular checking does model only one feature at a time, it still requires the presence of the base program and hence is not as modular as might be desired.

The essential points for our purposes are the elaboration on the idea of cleanly weakening of one feature with respect to another from Felty and Namjoshi [53], and identifying a plausible way to modularly specify and reason about open features. Moreover, their modularity result depends heavily on "unknown" values and on the cooperation of specification authors—oblivious properties cannot be modularly checked. Additionally, as presented, the authors acknowledge that they assume a linear ordering of feature composition, but state that the work easily extends to composing features in the absence of a base product. If so, the essence of this approach may be very useful for browser extensions, indicating what cooperation levels may be necessary for independent extension authors to produce easily composable extensions.

**Reified features** Plath and Ryan [102] propose making features an explicit construct of the language used to verify feature interactions. They also use a hybrid approach in which the system is specified as a state machine while the requirements over it are specified as temporal formulas. Together, these essentially form a feature aspect: a reified, syntactic component that may specify the presence of required data, functionality or other features; may introduce new data, functionality, or specifications; and may change the behavior of existing data. Like aspect weaving, this last step is inherently not modular. They identify four types of conflict between two extensions, reminiscent of aspect conflicts: a feature composed later (resp. earlier) breaks one composed earlier (resp. later); both features together break a property

[10]The actual algorithm is much subtler for improved precision; the gist shown here omits the requisite but confusing bookkeeping.

satisfied by the base system and either feature alone; finally, the two features do not commute.

Firefox extensions are already syntactically reified, declarative objects. (They are not quite first-class in the language sense, as they cannot be assigned as values.) Extending them to include verification information would be natural. The exact technical assumptions made in this work may be too expensive: the authors admit that even in their case study with a small number of simple features, the state-space explosion quickly made the verification intractable. Perhaps surprisingly, they argue for relative nonchalance over conflicts that break the original system (after all, extensions are intended to change system behavior). It is unclear whether this argument is warranted for browsers: most browser extensions do not supplant core functionality (unlike some telephony features examined here), but merely augment it.

# 6 Extensibility in Web Browsers

A browser is a multi-layered architecture: in its core are subsystems for rendering HTML and CSS, running Javascript, using the network and disk, and applying security policies to downloaded content. Beside these modules, nearly every browser supports a binary plugin API through which plugins can add support for new media formats and new interface elements to trigger their functionality. These plugins run in the browser process, and are unconstrained in their behavior beyond implementing a few key interfaces. They are analogous to kernel drivers in commodity OS designs: like drivers, a poorly-written plugin can destabilize the entire browser. In most browsers, only one plugin can be active for a given extension point (e.g., a media file type) at a given time, which neatly avoids any conflicts between plugins.

Like user-mode applications in an OS, web pages from different origins are isolated from one another. This level supports two closely related kinds of extensions: *bookmarklets* and *user scripts*. A bookmarklet is a bookmark containing a snippet of script that, when selected by a user, runs in the context of the current page. User scripts do the same, but are activated automatically upon navigating to a page that the script selects as relevant. These extensions are analogous to, and as powerful as, remote thread injection in operating systems. If multiple bookmarklets or user scripts run on a page, they may conflict with one another and break that web page, but cannot interact with or do any harm to the browser as a whole (assuming proper page isolation [104]). All modern browsers support bookmarklets; nearly all have support for user scripts either built into the browser or available as a plugin.

Between these two levels of the browser sits a third, which so far has been explored only in the Firefox browser[11]. Firefox defines most of the functionality and appearance of its user interface using XUL (a markup language much like HTML) and CSS styling, complete with a DOM (the document object model, which makes available to scripts the structure and behaviors of the currently displayed document), and additionally exposes JS bindings to internal functionality. In other words, UI and functionality in Firefox can be programmed *just like* web pages can, except

[11] Technically, Firefox also supports a fourth layer below all of these, which admits extensions to the Gecko runtime (e.g. debugging extensions such as Chromebug). We view this as a technical artifact, and not a necessary part of an extensible browser's design.

more powerfully. These languages are exposed to extensions, giving them the same flexibility as Firefox itself uses. Extensions run with the same privileges as the browser kernel (for instance, free of the same-origin policy) and can add, remove or modify UI and functionality. This type of extensibility has no exact analogue in operating systems.

For comparison, besides binary plugins, Internet Explorer 8 now supports two new forms of extension, Web Slices and Accelerators [76, 96, 97], which are quicker ways of interacting with web content. Analogues of both abilities have been implemented as Firefox extensions [52, 57, 98]. Both Accelerators and Web Slices offer limited extensibility: authors can define short XML files which add support for a new accelerator or slice; these extensions cannot interact with each other at all. Likewise, Opera has deliberately chosen not to include extension mechanisms in its product beyond binary plugins, opting instead to hard-code a wide variety of additional functionality into the browser itself [119]. They do support widgets, which are web-like bundles of HTML, CSS and JS code that run "on the desktop" outside the visual confines of a browser window. There is no meaningful notion of conflict between widgets: while they can read files written by other widgets, this is not inherently a widget-level conflict. [101, and articles].

Since Firefox extensions are intentionally programmed similarly to web pages, one might wonder whether web programming suffers similar extensibility conflicts. Certainly, heavily-scripted pages cannot simply be excerpted into other pages and be expected to work properly—the scripts and styles from the old page will conflict with the new one. Similarly, multiple JS libraries (such as jQuery and Dojo) cannot easily be combined on a single page any more than can, say, GTK and Qt in a desktop application. The closest web-page analogue to a browser with multiple extensions is a *mashup*: a web site that aggregates data from multiple sites to display something new. Three differences make this a less exciting challenge: first, a mashup author is a developer (unlike a browser user) and so has the skill to refine the mashup until it works. Second, mashups do not weave multiple sites onto each other (as extensions weave into the browser), but rather use scripts to extract the content from multiple sites and produce the mashup. This has led to the development of mashup editors such as Yahoo! Pipes and Open Mashups Studio; no such help exists for extension authors. Finally, unlike scripted web pages, Firefox extensions only target a single browser.

## 6.1 Firefox's extension model

In more detail, Firefox extensions consist of three primary pieces: a set of XUL overlays that define new UI content to be appended to existing UI that itself is defined in XUL, a set of JS files that define the functionality of the extension, and a simple manifest that declares some minimal metadata about the extension to Firefox. XUL, like HTML, exposes a DOM to scripts that manipulate it. Extensions can therefore define new UI with as rich capabilities as Firefox itself, and their scripts can manipulate both new and existing content. Additionally, Firefox exports a very wide API for internal functionality, which extensions can use to modify the browser's behavior.

Firefox extensions lie in a region of the design space distinct from the previous domains we have examined. First,

and perhaps obviously, Firefox extensions must define both functionality and user interfaces; all the examples we have seen so far have dealt primarily with functionality[12]. As such we have to be careful to distinguish UI-specific architectural choices from functionality ones, especially when discussing abilities of and conflicts among extensions.

Focusing first on design considerations, Firefox's extension mechanisms have been used to define new policies and new functionality for the browser. These extensions have been written primarily by non-Firefox developers, unlike most of the domains studied above where extension authors typically were also the mainline code's authors. Extension authors are given an erratic level of cooperation from Firefox, with strong support for UI extension and weaker cooperation for code extension. The UI extension mechanism only permits XUL elements with identifiers to be extended: on the one hand, this (like Aldritch's open modules) requires pervasive cooperation in Firefox to permit arbitrary extension; on the other, giving identifiers to (nearly all) page elements is standard practice in HTML and so is no onerous burden. Firefox's code extension, on the other hand, offers no cooperation to the extension author; indeed, because JS is so mutable at runtime, none is needed. UI extensions are integrated into Firefox at load-time using a simplified analogue of aspect weaving. Code extensions, however, may be integrated at any point after loading, since JS makes no essential distinction between load-time and runtime. In short, from design considerations alone, Firefox's extensions are closest to OS extensions in terms of authorship, integration time and cooperation from the mainline system.

Turning to extension abilities, Firefox extensions target a rich variety of resources: UI, code, web content, profile settings, and local storage, for example. All of these are accessed either declaratively (for UI) or via code (everything else). UI declarations are strictly additive, and apply on a per-element basis, which is the inherent granularity of the DOM. Their pervasiveness depends on how much of the existing UI has been annotated with identifiers; in practice, this is very pervasive. Code extensions are likewise granular but far more pervasive: the DOM permits scripts to remove and modify existing content (as well as dynamically add new elements), and JS permits the redefinition of existing code; together these permit scripts to modify almost anything. Consequently, extension interactions are difficult to describe.

Firefox permits users to install multiple extensions in a given profile simultaneously; extensions load in some unspecified order as Firefox loads. Interactions between extensions can occur via any of the resources mentioned above. Extensions may modify each other's UI, dynamically change each other's code, or interfere with each other's state. (These last two are particularly troublesome: since JS has no namespace mechanism for modularity, poorly-written extensions pollute the global namespace, which can silently redefine code added by other extensions.) Some of these interactions are intentional: for instance, TabMix Plus and Session Manager both implement saving and restoring of browser sessions (open windows and tabs), and the former changes both extensions' UI to enable exactly one or the other's ability, as both managers running simultaneously would break. However, many extensions have inadvertent conflicts: FoxyTunes inserts a small Flash object to play MP3 files directly from the browser, while FlashBlock eponymously disables this functionality. Other, more subtle interactions are possible, artifacts due to quirks in Firefox's

[12]Feature specification comes closest, albeit indirectly, to dealing with user interaction.

architecture. Thus from an extension abilities viewpoint, Firefox extensions are most like unrestrained aspects.

Firefox only implements very rudimentary conflict checking: an extension's manifest can declare with which versions of Firefox it is compatible, and which other extensions it depends upon for successful execution; these checks can be enforced at extension installation and load time. However, all interactions over UI, code or execution state are not detectable until runtime, by users who likely cannot debug problems they encounter. From a troubleshooting perspective, Firefox extensions ought to be designed slightly closer to static feature specification than runtime monitoring.

## 6.2 Improving the extension model

Firefox's extension model is not ideal, but it is far more capable than any other current browser extension model. Firefox's glaring failing is in conflict detection. Here we suggest an idealized extension model, and relate the techniques explored earlier to this new domain. Because extensions by nature split into functionality and interface, it is natural that the techniques to resolve UI conflicts be different than those tailored for functionality.

*An idealized extensible browser permits extensions to modify the state, functionality, and appearance of the browser in a fine-grained, pervasive manner. Extensions are written by amateur developers external to the browser, and should require minimal cooperation from the browser to run successfully. Interactions between extensions are possible: extensions may communicate, may adapt to each other's presence, and may modify the mainline browser in benign ways. Conflicts may occur when extensions try to modify each other or the mainline browser in incompatible ways, by changing UI or invariants upon which the other relies. It is not acceptable that extension conflicts first be found by end-users at runtime, so conflicts must be detectable by load-time and preferably at install time. Extensions must have a mechanism for controlling their composition to resolve conflicts, either automatically or through user intervention.*

Considering first UI conflicts, we may reasonably expect feature specification techniques to be readily applicable. Indeed, viewing sequences of user interaction as features, feature specification readily checks that patterns of interaction remain feasible despite the addition of new features. (For instance, a feature could declare that "from any state, the Save functionality is available" and "once the Save menu item or Save toolbar button is selected, the page will eventually be saved." Extensions may then remove either the toolbar or the menu and not violate the existing feature, but removing both would result in conflict.) Ideally, we would like conflict checking to be as efficient as possible (since users are notoriously impatient), which argues for modular verification of extensions. Unfortunately, modularity is in direct tension with our desire for minimal cooperation: while Li et al. [84, 85] enable modular checking of features compared to Felty and Namjoshi [53], it comes at the cost of explicitly specifying interaction points between features. Depending on how this is presented to the developer, this again may not be too onerous or restrictive a requirement. Additionally, Douence et al. [37] distinguish the notions of strong and weak independence. Adapting this notion to UI specification, we see that the more extensions can be certified as strongly independent (i.e., never in conflict regardless of other

extensions), the more modularly they can be verified for other feature interactions. In fact, in preliminary work we have modeled XUL overlays as document transformers; weak and strong independence fell out naturally from our definitions.

Turning to address conflicts from code as well, we see the full flexibility of Javascript is a liability, and some more constrained language is necessary to prevent conflicts here. This was the approach taken by SPIN [13] and Singularity [34], which both abandoned C/C++ for a strongly typed extension language that permitted compile-time assurances of correctness. The danger with static type systems is they are specialized to enforce a particular safety property of the code; as we have seen, not all policies are safety policies, and policies may override each other when composed. However as Singularity has shown, a strong type system may be sufficient (but not necessary) to enforce an over-approximation of desired properties, provided one uses a narrow interface with just the right abstractions. The engineering challenge in adapting Singularity's approach will be to apply a typing discipline to the DOM [122] and extend it to a suitably constrained interface to the browser's runtime services.

It is likely that many browser extensions cannot be considered "harmless," as they deliberately change behaviors of the browser. However, perhaps a weaker noninterference result holds: if the browser declares certain behaviors as extensible (à la open modules), harmless extensions merely must not modify anything else. If not: Javascript routinely employs event handlers that are dynamically added and removed; perhaps these patterns can be expressed using the dynamic pointcuts of Douence et al. [37, 38], or even more powerfully using Felty and Namjoshi's intuitive specification language [53]? Then the absence of conflicts using these systems would imply extension compatibility.

Not all conflicts may be resolvable by load-time; sometimes users may wish to choose to dynamically prioritize one extension over another. Here security monitors' runtime techniques become relevant. Assuming we can present the user with a sufficient approximation of all conflicts between two extensions, the user can define a policy resolving the conflicts. When more than two extensions conflict, we may naturally need to compose policies. Moreover, these need not be simple security policies: it is too limiting to ensure only that no bad thing occurs (e.g., two extensions activate simultaneously); we may want to ensure that eventually something good occurs (e.g., both extensions run in a well-defined order). For appropriately expressive sets of "security-relevant" events, this is precisely Polymer's aim [12].

# 7 Conclusion

This report has examined the extensibility problem: how systems should be designed to support extensions, and how we can reason about conflicts between them. Our target system has been the relatively new extensible web browser, and we have explored extensibility efforts in aspect-oriented programming, operating systems, feature specification and security monitors in order to better understand the browser. We developed a classification scheme through which we have defined the extensibility models for each of these systems. Finally, we have examined several specific techniques from these domains, and explored how they might be used to design a better extensibility model for a web browser.

# References


[1] M. Abadi and L. Lamport, "Open systems in tla," in *PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1994, pp. 81–90.

[2] ——, "Conjoining specifications," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 3, pp. 507–535, 1995.

[3] I. Aktug and K. Naliuka, "Conspec – a formal language for policy specification," *Electronic Notes in Theoretical Computer Science*, vol. 197, no. 1, pp. 45–58, 2008, proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007). [Online]. Available: http://www.sciencedirect.com/science/article/B75H1-4RWBP2W-5/2/fe9b06d3f2eaa797fb6bf579f4b25c93

[4] M. Al-Mansari, S. Hanenberg, and R. Unland, "On to formal semantics for path expression pointcuts," in *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2008, pp. 271–275.

[5] J. Aldrich, "Open modules: Modular reasoning about advice," in *ECOOP*, ser. Lecture Notes in Computer Science, vol. 3586. Springer Berlin / Heidelberg, 2005, pp. 144–168.

[6] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, no. 3, pp. 117–126, Sep. 1987.

[7] T. Anderson, "The case for application-specific operating systems," in *Proc. Third Workshop on Workstation Operating Systems*, 1992. [Online]. Available: http://www.cs.washington.edu/homes/tom/pubs/app-spec.html

[8] E. Artiaga, A. Serra, and M. Gil, "Porting multithreading libraries to an exokernel system," in *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*. New York, NY, USA: ACM, 2000, pp. 121–126.

[9] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview," *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, vol. 3362, pp. 49–69, 2005.

[10] L. Bauer, J. Ligatti, and D. Walker, "Types and effects for non-interfering program monitors," in *International Symposium on Software Security*, ser. Lecture Notes in Computer Science, M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, Eds., vol. 2609. Springer, 2002, pp. 154–171.

[11] ——, "Types and effects for non-interfering program monitors," *Software Security âĂŤ Theories and Systems*, vol. 2609, pp. 253–264, 2003.

[12] ——, "Composing security policies with polymer," *SIGPLAN Not.*, vol. 40, no. 6, pp. 305–314, 2005.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers, "Extensibility safety and performance in the SPIN operating system," in *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1995, pp. 267–283.

[14] K. Bierhoff and C. Hawblitzel, "Checking the hardware-software interface in spec#," in *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems*. New York, NY, USA: ACM, 2007, pp. 1–5.

[15] J. Bisbal and B. H. C. Cheng, "Resource-based approach to feature interaction in adaptive software," in *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. New York, NY, USA: ACM, 2004, pp. 23–27.

[16] L. Blair, T. Jones, and S. Reiff-Marganiec, "A feature manager approach to the analysis of component-interactions," in *FMOODS '02: Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems V*. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 2002, pp. 233–248. [Online]. Available: http://portal.acm.org/citation.cfm?id=771622

[17] A. Bolour. (2003, July) Notes on the eclipse plug-in architecture. [Online]. Available: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

[18] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin, "The feature interaction problem in telecommunications systems," in *Software Engineering for Telecommunication Switching Systems, 1989. SETSS 89., Seventh International Conference on*, Jul 1989, pp. 59–62. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=41848

[19] C. Breuel and F. Reverbel, "Join point selectors," in *SPLAT '07: Proceedings of the 5th workshop on Software engineering properties of languages and aspect technologies*. New York, NY, USA: ACM, 2007, p. 3.

[20] G. Bruns and P. Godefroid, "Generalized model checking: Reasoning about partial state spaces," in *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory*. London, UK: Springer-Verlag, 2000, pp. 168–182.

[21] M. Cain, "Managing run-time interactions between call-processing features [intelligent networks]," *Communications Magazine, IEEE*, vol. 30, no. 2, pp. 44–50, Feb 1992. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=127557

[22] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Computer Networks*, vol. 41, no. 1, pp. 115 – 141, 2003.

[23] M. Chechik, S. Easterbrook, and B. Devereux, "Model checking with multi-valued temporal logics," *Multiple-Valued Logic, 2001. Proceedings. 31st IEEE International Symposium on*, vol. 0, pp. 187–192, 2001.

[24] C. Clifton and G. T. Leavens, "Observers and assistants: A proposal for modular aspect-oriented reasoning," in *Foundations of Aspect Languages*, 2002, pp. 33–44.

[25] T. Colcombet and P. Fradet, "Enforcing trace properties by program transformation," in *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2000, pp. 54–66.

[26] P. Costanza, "Dynamically scoped functions as the essence of aop," *SIGPLAN Not.*, vol. 38, no. 8, pp. 29–36, 2003.

[27] D. S. Dantas and D. Walker, "Harmless advice," in *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2006, pp. 383–396.

[28] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich, "PolyAML: a polymorphic aspect-oriented functional programming language," in *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2005, pp. 306–319.

[29] ——, "AspectML: A polymorphic aspect-oriented functional programming language," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 1–60, 2008.

[30] B. De Fraine, M. Südholt, and V. Jonckers, "StrongAspectJ: flexible and safe pointcut/advice bindings," in *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2008, pp. 60–71.

[31] A. S. de Oliveira, "Rewriting-based access control policies," *Electronic Notes in Theoretical Computer Science*, vol. 171, no. 4, pp. 59–72, 2007, proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006). [Online]. Available: http://www.sciencedirect.com/science/article/B75H1-4P30GP3-5/2/5e55d586fb30e85ac887a3cc4805452f

[32] A. S. de Oliveira, E. K. Wang, C. Kirchner, and H. Kirchner, "Weaving rewrite-based access control policies," in *FMSE '07: Proceedings of the 2007 ACM workshop on Formal methods in security engineering*. New York, NY, USA: ACM, 2007, pp. 71–80.

[33] G. Denys, F. Piessens, and F. Matthijs, "A survey of customizability in operating systems research," *ACM Comput. Surv.*, vol. 34, no. 4, pp. 450–468, 2002.

[34] J. DeTreville, "Making system configuration more declarative," in *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2005, pp. 11–11. [Online]. Available: http://www.usenix.org/events/hotos05/final_papers/detreville.html

[35] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, "Leveraging legacy code to deploy desktop applications on the web." in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 339–354. [Online]. Available: http://research.microsoft.com/apps/pubs/?id=72878

[36] R. Douence, O. Motelet, and M. Südholt, "A formal definition of crosscuts," in *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. London, UK: Springer-Verlag, 2001, pp. 170–186.

[37] R. Douence, P. Fradet, and M. Südholt, "A framework for the detection and resolution of aspect interactions," in *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*. London, UK: Springer-Verlag, 2002, pp. 173–188.

[38] ——, "Composition, reuse and interaction analysis of stateful aspects," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2004, pp. 141–150.

[39] L. du Bousquet, F. Ouabdesselam, J. L. Richier, and N. Zuanon, "Feature interaction detection using a synchronous approach and testing," *Computer Networks*, vol. 32, no. 4, pp. 419 – 431, 2000.

[40] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi, "Semantics and scoping of aspects in higher-order languages," *Sci. Comput. Program.*, vol. 63, no. 3, pp. 207–239, 2006.

[41] R. Echahed and F. Prost, "Security policy in a declarative style," in *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. New York, NY, USA: ACM, 2005, pp. 153–163.

[42] A. Edwards and G. Heiser, "Components + security = os extensibility," *Aust. Comput. Sci. Commun.*, vol. 23, no. 4, pp. 27–34, 2001.

[43] E. A. Emerson and J. Y. Halpern, ""sometimes" and "not never" revisited: on branching versus linear time temporal logic," *J. ACM*, vol. 33, no. 1, pp. 151–178, 1986.

[44] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*. Washington, DC, USA: IEEE Computer Society, 1995, p. 78.

[45] D. R. Engler, M. F. Kaashoek, and J. O. Jr., "Exokernel: an operating system architecture for application-level resource management," in *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1995, pp. 251–266.

[46] Ú. Erlingsson, "The inlined reference monitor approach to security policy enforcement," Ph.D. dissertation, Cornell University, Ithaca, NY, USA, 2004, adviser-Schneider, Fred B. [Online]. Available: http://portal.acm.org/citation.cfm?id=997617

[47] Ú. Erlingsson and F. B. Schneider, "Sasi enforcement of security policies: a retrospective," in *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*. New York, NY, USA: ACM, 2000, pp. 87–95.

[48] ——, "Irm enforcement of java stack inspection," in *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2000, p. 246.

[49] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *OSDI*, 2006. [Online]. Available: http://research.microsoft.com/research/sv/gleipnir/

[50] D. Evans and A. Twyman, "Flexible policy-directed code safety," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999, pp. 32–45. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=766716

[51] D. Evans, "Policy-drected code safety," Ph.D. dissertation, Massachusetts Institute of Technology, 1999. [Online]. Available: http://www.cs.virginia.edu/~evans/phd-thesis/abstract.html

[52] A. Faaborg. (2006) Introducing operator. [Online]. Available: http://labs.mozilla.com/2006/12/introducing-operator/

[53] A. P. Felty and K. S. Namjoshi, "Feature specification and automated conflict detection," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 1, pp. 3–27, 2003.

[54] R. E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," Research Institude for Advanced Computer Science, Tech. Rep., 2000.

[55] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker, "patch (1) considered harmful," in *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2005, pp. 16–16.

[56] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Brice no, R. Hunt, and T. Pinckney, "Fast and flexible application-level networking on exokernel systems," *ACM Trans. Comput. Syst.*, vol. 20, no. 1, pp. 49–83, 2002.

[57] D. Glazman. (2009) Search – webchunks. [Online]. Available: http://www.glazman.org/weblog/dotclear/index.php?q=webchunks

[58] R. Grimm and B. N. Bershad, "Separating access control policy, enforcement, and functionality in extensible systems," *ACM Trans. Comput. Syst.*, vol. 19, no. 1, pp. 36–70, 2001.

[59] K. W. Hamlen and M. Jones, "Aspect-oriented in-lined reference monitors," in *PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*. New York, NY, USA: ACM, 2008, pp. 11–20.

[60] P. B. Hansen, "The nucleus of a multiprogramming system," *Commun. ACM*, vol. 13, no. 4, pp. 238–241, 1970.

[61] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit, "A graph-based approach to modeling and detecting composition conflicts related to introductions," in *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2007, pp. 85–95.

[62] J. D. Hay and J. M. Atlee, "Composing features and resolving interactions," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 6, pp. 110–119, 2000.

[63] C. Hofer and K. Ostermann, "On the relation of aspects and monads," in *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*. New York, NY, USA: ACM, 2007, pp. 27–33.

[64] P. Hui and J. Riely, "Typing for a minimal aspect language: preliminary report," in *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*. New York, NY, USA: ACM, 2007, pp. 15–22.

[65] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber, "Sealing os processes to improve dependability and safety," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 341–354.

[66] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, 2007.

[67] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber, "Broad new os research: challenges and opportunities," in *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2005, pp. 15–15. [Online]. Available: http://www.usenix.org/events/hotos05/final_papers/hunt.html

[68] A. Igarashi and N. Kobayashi, "Resource usage analysis," in *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2002, pp. 331–342.

[69] F. Jacquemard, M. Rusinowitch, and L. Vigneron, "Compiling and verifying security protocols," *Logic for Programming and Automated Reasoning*, vol. 1955, pp. 535–554, 2000.

[70] R. Jagadeesan, A. Jeffrey, and J. Riely, "A calculus of untyped aspect-oriented programs," in *European Conference on Object-Oriented Programming*. Springer-Verlag, 2003, pp. 54–73.

[71] R. Jagadeesan, C. Pitcher, and J. Riely, "Open bisimulation for aspects," in *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2007, pp. 107–120.

[72] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. B. no, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie, "Application performance and flexibility on exokernel systems," in *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1997, pp. 52–65.

[73] E. Katz and S. Katz, "Incremental analysis of interference among aspects," in *FOAL '08: Proceedings of the 7th workshop on Foundations of aspect-oriented languages*. New York, NY, USA: ACM, 2008, pp. 29–38.

[74] D. O. Keck and P. J. Kuehn, "The feature and service interaction problem in telecommunications systems: A survey," *IEEE Trans. Softw. Eng.*, vol. 24, no. 10, pp. 779–796, 1998.

[75] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2001, pp. 327–353.

[76] J. Kim. (2008) Activities and webslices in internet explorer 8. [Online]. Available: http://blogs.msdn.com/ie/archive/2008/03/06/activities-and-webslices-in-internet-explorer-8.aspx

[77] S. Kojarski and D. H. Lorenz, "Pluggable aop: designing aspect mechanisms for third-party composition," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 247–263.

[78] ——, "Identifying feature interactions in multi-language aspect-oriented frameworks," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 147–157.

[79] ——, "Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions," *SIGPLAN Not.*, vol. 42, no. 10, pp. 515–534, 2007.

[80] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, 1994199.

[81] B. Lampson, "On reliable and extendible operating systems," in *Proc. 2nd NATO Conf. on Techniques in Software Engineering*, 1971, pp. 421–444. [Online]. Available: https://research.microsoft.com/en-us/um/people/blampson/07-ReliableOS/07-ReliableOSAbstract.htm

[82] T. Leschke, "Achieving speed and flexibility by separating management from protection: embracing the exokernel operating system," *SIGOPS Oper. Syst. Rev.*, vol. 38, no. 4, pp. 5–19, 2004.

[83] N. Lesiecki. (2005, May) AOP@Work: Enhance design patterns with AspectJ, part 1. [Online]. Available: http://www.ibm.com/developerworks/java/library/j-aopwork5/index.html

[84] H. Li, S. Krishnamurthi, and K. Fisler, "Verifying cross-cutting features as open systems," in *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2002, pp. 89–98.

[85] H. C. Li, S. Krishnamurthi, and K. Fisler, "Modular verification of open features using three-valued model checking," *Automated Software Eng.*, vol. 12, no. 3, pp. 349–382, 2005.

[86] J. Liedtke, "On micro-kernel construction," in *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1995, pp. 237–250.

[87] ——, "Toward real microkernels," *Commun. ACM*, vol. 39, no. 9, pp. 70–77, 1996.

[88] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: enforcement mechanisms for run-time security policies," *Int. J. Inf. Sec*, vol. 4, no. 1-2, pp. 2–16, Feb. 2005.

[89] ——, "Enforcing non-safety security policies with program monitors," in *Computer Security—ESORICS 2005: 10th European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, vol. 3679, Sep. 2005, pp. 355–373.

[90] J. Ligatti, D. Walker, and S. Zdancewic, "A type-theoretic interpretation of pointcuts and advice," *Sci. Comput. Program.*, vol. 63, no. 3, pp. 240–266, 2006.

[91] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of nonsafety policies," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 3, pp. 1–41, 2009.

[92] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, and W. Schröder-Preikschat, "Configurable memory protection by aspects," in *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems*. New York, NY, USA: ACM, 2007, pp. 1–5.

[93] P. A. Loscocco and S. D. Smalley, "Meeting critical security objectives with security-enhanced linux," in *Proceedings of the 2001 Ottawa Linux Symposium*, 2001. [Online]. Available: http://www.nsa.gov/research/_files/selinux/papers/ottawa01-abs.shtml

[94] H. Masuhara, H. Tatsuzawa, and A. Yonezawa, "Aspectual caml: an aspect-oriented functional language," in *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2005, pp. 320–330.

[95] A. Metzger, "Feature interactions in embedded control systems," *Comput. Netw.*, vol. 45, no. 5, pp. 625–644, 2004.

[96] Microsoft. (2009) Add-ons gallery. [Online]. Available: http://www.ieaddons.com/en/

[97] Microsoft Developer Network. (2009) Browser extensions. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa753587(VS.85).aspx

[98] A. Mohta. (2009) Get IE8 accelerator in firefox : Select n go. [Online]. Available: http://www.technospot.net/blogs/get-ie-8-accelerators-in-firefox/

[99] G. Muller, Y. Padioleau, J. L. Lawall, and R. R. Hansen, "Semantic patches considered helpful," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, pp. 90–92, 2006.

[100] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam, "Adding open modules to AspectJ," in *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2006, pp. 39–50.

[101] Opera. (2009) Widgets. [Online]. Available: http://dev.opera.com/articles/widgets/

[102] M. Plath and M. Ryan, "Feature integration using a feature construct," *Sci. Comput. Program.*, vol. 41, no. 1, pp. 53–84, 2001.

[103] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1989, pp. 179–190.

[104] C. Reis and S. D. Gribble, "Isolating web programs in modern browser architectures," in *Eurosys*, 2009.

[105] C. Reis, S. D. Gribble, and H. M. Levy, "Architectural principles for safe web programs," in *Proceedings of the Sixth Workshop on Hot Topics in Networks (HotNets)*, Nov 2007.

[106] A. Reynolds, M. E. Fiuczynski, and R. Grimm, "On the feasibility of an aosd approach to linux kernel extensions," in *ACP4IS '08: Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*. New York, NY, USA: ACM, 2008, pp. 1–7.

[107] M. Rinard, A. Salcianu, and S. Bugrara, "A classification system and analysis for aspect-oriented programs," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 147–158, 2004.

[108] C. Rippert, "Protection in flexible operating system architectures," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 4, pp. 8–18, 2003.

[109] Y. Saito and B. Bershad, "A transactional memory service in an extensible operating system," in *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1998, pp. 5–5.

[110] S. B. Sanjabi and C.-H. L. Ong, "Fully abstract semantics of additive aspects by translation," in *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2007, pp. 135–148.

[111] S. Savage and B. N. Bershad, "Issues in the design of an extensible operating system," in *OSDI*, 1994, p. 196. [Online]. Available: http://www-spin.cs.washington.edu/papers/index.html

[112] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.

[113] D. Sereni and O. de Moor, "Static analysis of aspects," in *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2003, pp. 30–39.

[114] S. Siddiqi and J. M. Atlee, "A hybrid model for specifying features and detecting interactions," *Comput. Netw.*, vol. 32, no. 4, pp. 471–485, 2000.

[115] C. Skalka and S. Smith, "Static enforcement of security with types," *SIGPLAN Not.*, vol. 35, no. 9, pp. 34–45, 2000.

[116] C. Small and M. Seltzer, "A comparison of os extension technologies," in *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1996, pp. 4–4.

[117] D. R. Smith, "Requirement enforcement by transformation automata," in *FOAL '07: Proceedings of the 6th workshop on Foundations of aspect-oriented languages*. New York, NY, USA: ACM, 2007, pp. 5–14.

[118] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi, "Solving the starting problem: device drivers as self-describing artifacts," in *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. New York, NY, USA: ACM, 2006, pp. 45–57.

[119] K. Thakker. (2008) Opera CEO: Why no extensions (so far) in opera. [Online]. Available: http://my.opera.com/kamalesh/blog/2008/04/19/why-no-extensions-so-far-in-opera-from-the-ceo

[120] The AspectJ Team. (2005) The AspectJ 5 development kit developer's notebook. [Online]. Available: http://www.eclipse.org/aspectj/doc/next/adk15notebook/

[121] ——. (2003) The AspectJ programming guide. [Online]. Available: http://www.eclipse.org/aspectj/doc/released/progguide/index.html

[122] P. Thiemann, "A type safe dom api," *Database Programming Languages*, vol. 3774, pp. 169–183, 2005.

[123] D. B. Tucker and S. Krishnamurthi, "Pointcuts and advice in higher-order languages," in *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2003, pp. 158–167.

[124] M. Vardi, "Sometimes and not never re-revisited: on branching versus linear time," *CONCUR'98 Concurrency Theory*, vol. 1466, pp. 1–17, 1998.

[125] M. Viswanathan, "Foundations for the run-time analysis of software systems," Ph.D. dissertation, University of Pennsylvania, Philadelphia, PA, USA, 2000.

[126] D. Walker, S. Zdancewic, and J. Ligatti, "A theory of aspects," in *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2003, pp. 127–139.

[127] M. Wang, K. Chen, and S.-C. Khoo, "Type-directed weaving of aspects for higher-order functional languages," in *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. New York, NY, USA: ACM, 2006, pp. 78–87.

[128] G. Washburn and S. Weirich, "Good advice for type-directed programming: aspect-oriented programming and extensible generic functions," in *WGP '06: Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*. New York, NY, USA: ACM, 2006, pp. 33–44.

[129] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: the kernel of a multiprocessor operating system," *Commun. ACM*, vol. 17, no. 6, pp. 337–345, 1974.