Practical system-level adaptation of heterogeneous components

a Thesis Proposal

Stephen Kell

January 2008

1 Introduction

As software systems become more complex, it becomes economically desirable to re-use existing pieces (or *components*) of software. Similarly, with the expanding space of applications, together with the ever-increasing ubiquity and mobility of devices, there is an ever-greater need for software which can be composed in novel and unanticipated ways, by users, at run time. For these practices to become widespread, they must be feasible in cases where the components have been developed independently of one another. In such cases, software interfaces are almost certain to be *mismatched* in some way, meaning that they cannot be correctly composed directly. To compose such mismatched components, therefore, some kind of *adaptation* is necessary.

Current development practices are far from optimal in their predispositions towards re-use and composition. This proposal concerns research into new ways by which existing software artifacts may be combined, and by which new software artifacts may be written so as later to be more amenable to adaptation and re-use. The key to these improvements is effecting a separation of concerns, between code which implements functionality and code which implements integration (i.e. communication). In the following sections I will motivate and outline the design of a system which enables and promotes this separation.

2 Motivation

When pieces of software are developed independently, yet have logically compatible functionalities, making them work together is often non-trivial. There are effectively two variants of this problem: firstly, incorporating a selection of re-used artifacts amid a novel environment, which can be tailored to support those artifacts; secondly, combining multiple re-used artifacts together more-or-less directly, such that additional code would ideally be unnecessary. I will call these the "re-usability" and "re-use" problems respectively. Both are interesting, but I will focus on the latter.

2.1 Open-source development

Perhaps the best evidence of the problem comes from world of open-source software. Even when source-code is fully and freely available, we anecdotally observe two phenomena. Firstly, most large programs' request-trackers contain requests for features which are already implemented in some other similar open-source project. This indicates that the effort required to port existing code for some logically compatible functionality is frequently non-trivial. Secondly, functionality is frequently duplicated in programs whose only distinction is in incidental implementation details irrelevant to their functionality: the choice of operating system, windowing toolkit, desktop environment, programming language, network protocol, storage abstraction, and so on. A major motivation for such duplication is invariably that the added homogeneity aids integration with other software.

As further illustration, here are some simple compositional use-cases which are currently impracticable without considerable coding effort.

- sharing bookmarks or history logs across multiple web browsers, or with other classes of application;
- adding a button to invoke a web-based natural language translator from within an e-mail client;
- writing a script (e.g. a `make` rule) which invokes the Postscript generator of an interactive graphical document editor;
- sharing a calendar between multiple applications concurrently, each notifying others of updates;
- pausing a media player application whenever another process requests the sound device.

Although we can imagine what code we might write to solve each problem individually, my concern is to invent the necessary supporting tools and runtime services to make the *entire class* of problem significantly more tractable. This means it should be *cheaper* for developers and *easier* for users to solve most common problems concerning communication between mismatched code. I do not expect these tasks to become fully automated nor, necessarily, trivial.

Similarly, although we can imagine how to implement a solution to each problem *for a particular codebase* (e.g. for Firefox, or for Emacs), my concern is rather to enable the developer to implement the solution only once, or at worst a small number of times, and have that implementation be cheaply composable with the large number of external codebases for which that feature is semantically meaningful.

(A final example might be the advertisement for Apple's iPhone product, showing in cinemas at the time of writing. A user begins with an instant messaging application, suggesting a cinema trip to a friend. He or she then uses a web browser to browse film descriptions, and finds candidate cinemas using a

mapping application backed by a web search engine. Finally, the user clicks a cinema on the map, which passes its phone number to the dialler application. Communicating these units of application-level meaning between continuously running applications currently requires considerable integration effort specific to the applications being combined.)

2.2 The need for adaptation

Adaptation is synonymous with resolving mismatch. Given our requirement that composition may be specified dynamically by end users, I propose a form of adaptation which is performed most often at load-time or run-time, is applicable to a wide range of existing software, and is intended to support evolutionary adoption. In the following sections we will overview the nature of adaptation, sketch a design of the system, and outline a plan for its implementation and evaluation.

3 What is adaptation?

I define adaptation as any process which modifies the form or behaviour of a subsystem to enable or improve communication with the surrounding parts of the system (which I call its *environment*). Adaptation may take many forms. It may be done *ahead-of-time*, at *load time* or at *run time*, may be *invasive* (i.e. modifying target code) or *non-invasive* (i.e. supplementing or interposing additional code)[1], *automatic* or *manual*, *binary-* or *source-level*, and may be done for *correctness* (i.e. the very ability to compose functional units of software) or for *optimisation*.

3.1 What needs adapting?

I assume familiarity with the problem space of adaptation. Unfamiliar readers should refer to Appendix A.

3.2 Existing practices

The problem of adaptation, while often not explicitly acknowledged by name, is certainly not a new one. Aside from writing "glue code" in conventional languages, many other established practices have particular relevance: scripting languages, service-oriented computing, aspect-oriented programming, interface definition languages, automatic marshalling in component middleware, configuration languages (e.g. in "inversion of control" frameworks), unified programming interfaces (e.g. Unix's "everything is a file"), unified binary interface (e.g. Microsoft's Common Language Runtime) and code metadata. None of these

[1] I use *invasive* and *non-invasive* synonymously with *white-box* and *black-box* respectively.

techniques is specifically designed to tackle the problem of mismatched programming interfaces. For more detailed consideration of each technique, see Appendix B.

4 The idea

I propose to investigate the thesis that "the complexity of composing heterogeneous mismatched components can be substantially reduced by adaptation abstractions which enable the separation of functionality from integration". To do so, I will devise and implement a linkage model which enables and encourages such separation, including a linking language containing adaptation features, and implementations of both static and dynamic linking. This amounts to a form of *non-invasive manual binary* adaptation.

The observations informing my approach are summarised in the following sections.

4.1 Separating functionality from integration

Code frequently incorporates knowledge about *how*, *with whom* and *using what conventions* it is to communicate. The avoidance of inlining such details is precisely the established good coding practice of "low coupling". However, even with the greater programmer discipline and foresight, it is simply not possible to communicate without assuming *some* details of communication. The essential feature of my approach is therefore to provide not only a separate domain, distinct from the programming language, in which to specify integration details, but also convenient ways to work around such details *from the outside* when mismatch does occur. The separate domain is a configuration language, and these "convenient ways" are adaptation primitives.

4.2 Hierarchical configuration

Some configuration languages, such as Darwin [**?**], have an explicit hierarchical structure, whereas others such as Reo [**?**] have a more general graph structure. Although hierarchy may seem an unnecessary restriction, it mirrors both human problem-solving and the recursive nature of the re-use paradigm—where new artifacts are created, recursively, as combinations of pre-existing and re-used ones. Providing a logical containment hierarchy might also tend to delineate those pieces of a system which turn out later to be convenient units of re-use.

4.3 Pragmatism

It is essential that we support adaptation of existing code, since the benefits of re-use are negated if the wealth of existing code cannot be exploited. Also, as with most outputs of research, the potential for impact is much greater if a technology can be adopted in an evolutionary fashion. Therefore, we must

support multiple code representations and languages, preferably in an extensible manner, and prioritise the support of popular languages (including C and Java).

Another pragmatic distinction comes from scripting languages. I have mentioned that hierarchy is useful for expressing logical groupings and hence aiding re-use. Notwithstanding this, some adaptations are too small to be practically re-usable. A benefit of scripting languages is their brevity, which makes them suitable for *invasive* adaptation, i.e. for altering code which is frequently changed [**?**]—perhaps during rapid prototyping, or for end-user customisation. In cases where adaptation logic is not complex enough to justify re-use, we must instead focus on making them brief to express in-line (as "ad-hoc" adaptation), and hence convenient to change. To do so, my configuration language must aim for brevity and expressivity comparable to that of scripting languages.

Finally, we observe that different object code representations contain differing levels of metadata. If we are to support many of these representations, including many popular ones, we must support those which provide little type information or other semantic annotations. It follows that we will not be able to guarantee safety of the compositions generated, unless their constituents happen to provide the necessary annotations—which we will not mandate. In other words, my system will value composability over safety or other property-checking. (The addition of pluggable property checking is discussed in Section 8.)

4.4 Extensible adaptation primitives

Rather than providing a fixed set of adaptation primitives, as with systems such as Nimble [**?**], I propose that this set should be extensible, i.e. that new adaptation primitives must be definable outside of the configuration language.

One important class of externally-definable primitives is that of generated adaptation. Much adaptation can be captured re-usably as adaptor generation algorithms, where a one-size-fits-all adaptor implementation would be inefficient. The inputs to such algorithms are the target pieces of code or their interface descriptions, perhaps supplemented with additional semantic specification. The output is the required adaptor code. Examples include adaptor synthesis algorithms [**?** **?** **?**], wrapper generators [**?**], IDL compilers, and convenient constructors for translation tables and parsers. It is crucial to have convenient support for invoking these generative adapters from within the linking language; they may be seen as adaptation functions, ranging over units of linkage.

5 Implementation

I intend to implement a configuration language, first as a static linking tool and second as an extension to the dynamic loader of a conventional modern operating system such as GNU/Linux. Specifically, I will devise and implement the following.

Configuration language A configuration language should be defined, which is capable of expressing adaptation over binary component representations. To support heterogeneity, the set of representations should be extensible. Likewise, the set of adaptation primitives must be extensible, meaning that new primitives may be added *outside* the language (analogously to how installing new Unix commands extends the language of the shell). A unifying model of components, linkage and adaptation must necessarily underpin the language's design.

Static implementation The language should be implemented, for ahead-of-time use, as a linker supporting adaptation. The linker should accept the configuration language, and generate fully-linked executables out of pre-existing object files and the like.

Dynamic implementation The language should be implemented, for run-time use, as a dynamic loader (in the sense of the C library's `dlopen()` et al) supporting adaptation. The loader must accept a variant of the configuration language, describing the components and adaptations to load.

Refactoring engine To complement the linking language, and demonstrate an alternative application of the separation between integration and functionality, a refactoring engine should be implemented. This will semi-automatically (i.e. with user assistance) refactor single source files in some popular existing language (either C or Java) into two refactored files: "core functionality", which will remain in the target language, and "integration logic", which will be captured in the configuration language.

5.1 Illustration

To illustrate the linking language, consider a toy system composed of two components, one or which is written to generate data as a series of tuples, and the other which expects to read data as a stream. Figure 1 shows a configuration which might implement such a system.

The top-level configuration combines two components, one which outputs tuples of the form (*sequence_no*, *bit_string*), and the other wishing to read a bit-stream. Such an arrangement might be used to handle out-of-order packet delivery in a network. The components are connected by a third component, which is a configuration of some ad-hoc adaptation and two smaller components: a tuple store and a re-usable adaptation component implementing a `take_contiguous(n)` procedure for the tuple space. The `take_contiguous` procedure retrieves a list of two-tuples with sequential sequence numbers, whose bit-strings do not exceed `n` in combined length. The ad-hoc adaptation projects out the bit-strings and flattens the resulting structure into a single string, which is handed to the stream reader as the output of a *read* call. Thick black lines indicate connector bindings.

Some plausible code for the example system is shown in Figure 2. The syntax is similar to that of Knit [**?**], a tool which allows precise specification of linkage

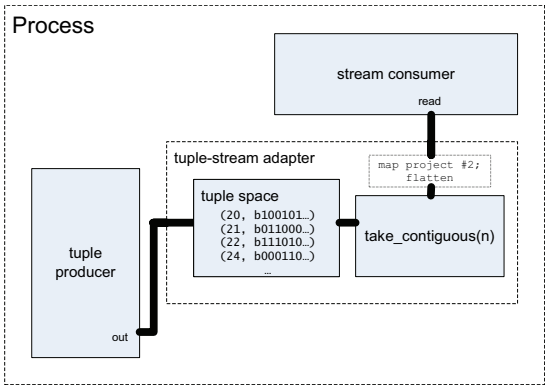

Figure 1: An example configuration

graphs under the traditional C linkage model. Note the following features in the code, referring back to the system diagram in Figure 1:

- the subdivision of each imported and exported interface into "roles", such as `prod` and `cons`;
- the differences in names of compatible roles (e.g. `cons` versus `dest` and the explicit mapping between the two using the *<import> <- <export>* syntax;
- the inline (or "ad-hoc") use of adaptation primitives `map` and `project`, respectively for function application over a list and for tuple element selection;
- the use of various constructors for different binary code representations, here showing `obj_elf("C", ...)` for an ELF object file with C linkage and `exec_elf(...)` for an ELF executable.

5.2 Novelty

I believe that the following contributions of the proposed work will be substantially novel:

- support for adaptation in a linking language;
- the pragmatic distinction between *ad-hoc* and *re-usable* adaptation;
- support for an *extensible* set of adaptation primitives in a configuration language;
- extension of an existing operating system's dynamic loading interface for implementing adaptation;

```
/* This syntax is simplified from Knit, with added adaptation features. Reserved
 * words are in bold, and denote either syntactic block kinds ("unit" or "compound"),
 * compositional operators ("<-" and "as") or built-in types. Identifiers refer either
 * to logical modules, interfaces (i.e. "roles"), linkage standards, symbols or files. */

unit myTupSpc {
        exports [ prod { (int, bit list) in_ordered() },
                  cons { void        out(int, bit list) } ];
        files { obj_elf("C", tuplespace.o) }
}

unit myTupleProducer {
        imports [ dest { void output(bit list, int) } ];
        exports [ /* ... */ ];
        files { obj_elf("C", tupleprod.o) }
}

unit myStreamConsumer {
        imports [ streamProvider { int read(byte addr, int) } ];
        exports [ /* ... */ ];
        files { obj_elf("C", streamcons.o) }
}

unit takeContiguous {
        imports [ source { (int, bit list) in_monotonic() } ];
        exports [ listProvider { (int, bit list) list get() } ];
        files { obj_elf("C", take_contig.o) }
}

/* The stanzas above are simply declarations for existing object files, and could
 * have been autogenerated from source. The next two blocks link them, using the
 * wiring operator '<-' and adaptation primitives 'map', 'project' and 'flatten'. */

compound tupleStreamAdapter {
        exports [ tupin  { void  out(int, bit list) },
                  strout { int   read(byte addr, int) } ];

        link obj_elf("C", "tupstream.o") [
                myTupSpc, takeContiguous
                ]
        {
                takeContiguous.source <- myTupSpc.prod { in_monotonic <- in_ordered }
                tupin as myTupSpc.cons;
                strout as takeContiguous.listProvider {
                        read as flatten(map (project #2) get)
                }
        }
}

compound Process {
        exports [ /* ... */ ];
        link exec_elf("process") [
                myTupleProducer, tupleStreamAdapter, myStreamConsumer
                ]
        {
                myTupleProducer.dest <- tupleStreamAdapter.tupin {
                        output(a, b) <- out(b, a)
                }
                myStreamConsumer.streamProvider <- tupleStreamAdapter.strout
        }
}
```

Figure 2: Example code for the configuration shown in Figure 1.

- refactoring to separate integration from functionality.

In addition, certain differences of approach or emphasis differentiate my proposal from existing work. These include the very strong practical emphasis, an insistence that the system will be useful for a wide variety of highly heterogeneous components, and a preference for composability ahead of safety or verifiability.

5.3 Practical approach

5.3.1 Static implementation and basic case-studies

The static linker implementation will be based on Knit [**?**], in the first instance, and the process of extending Knit will be used to refine out a suitable linkage model, notation, and set of basic adaptation primitives. This may or may not lead to a complete re-design which dispenses with all Knit implementation.

Case-study evaluation of this implementation will use well-known open-source codebases including Mozilla Firefox. Once the codebase's linkage relation has been captured in Knit code (generated mechanically), we may extract subgraphs corresponding to individual features and, using the adaptation features, integrate them with a separate codebase. For example, we might try extracting the browser history feature from Firefox and integrating it into a file manager. The primary success criterion is that this should be possible entirely from the linkage domain, without changing any existing source code of the file manager, and without introducing any adaptation primitives which would not be widely re-usable. From the experience gained during this work, I will develop a library of common adaptations, utilising the support for an extensible set of convenient adaptation primitives described in Section 4.4.

5.3.2 Dynamic implementation

Dynamic loading is a logical extension to the static linking case, and is essential for dynamic user-directed composition. Dynamic loading is a highly general mechanism, which can add arbitrary new code into the running process. This loaded code might function entirely within the current process, or it might be stub code for communication with external processes (or the kernel). As such, dynamic loading may serve as the unique mechanism by which any new communication channel is defined or brought into scope. For example, one could imagine rewriting the traditional C code

```
FILE *fp = fopen("/path/to/myfile", "r");
```

as a call to dynamically load a "file handle object", e.g.

```
FILE *fp = (FILE*) dlopen("/path/to/myfile!r", 0);
```

where we have unified file control block pointers with library handles, and encoded the read-only interface signifier `"r"` into the object name.

Note that this transfers perfectly well to languages providing safety guarantees. Run-time safety can be achieved by including an extra "type argument", e.g. in Java

```
T t = System.dlopen("/path/to/myfile!r", 0, T.class);
```

where `System.dlopen()` is polymorphic in `T` and throws an exception if the object denoted by `/path/to/myfile!r` doesn't satisfy type `T`. The type argument is an assertion about the type of the object which the `dlopen()` call is expected to return. Static type-safety requires additionally that the type encoded by the assertion can be inferred at compile-time, such as in the above Java example where `T.class`, having type `Class<T>`, allows the compiler to infer that the result will have type `T`.

I will implement the dynamic case by embedding a variant of the adaptation language into Linux's dynamic loader, and providing bindings for some common languages (including C and Java). Using this, and with help from the library, users will be able to dynamically specify extensions to their running applications by combining generic third-party code with adaptation logic. For example, a user might construct a browser plugin for natural language translation, dynamically, by specifying some adaptation which combines a local HTML parser library with a plain-text natural language translation web service. This, and other case-studies similar to the static case, will be sufficient for basic proof-of-concept.

To demonstrate the unifying power of dynamic loading, I will also reimplement the Unix filesystem and sockets interfaces as thin layers over the dynamic loader. As a result, programs written against only one or other of these interfaces (e.g. a webmail-to-POP gateway, binding to a socket for input) will be made to run just as easily against other targets (e.g. reading input from a log file containing a previous POP session) simply by adapting the socket address (e.g. to use a special address family, whose address structure can embed some adaptation expression denoting the session log file).

(It is possible to unify more than just the socket and filesystem interfaces. Note that `dlopen()` is essentially performing instantiation of *objects*—such objects are traditionally libraries, but might be finer-grained. Language-level object instantiation mechanisms could therefore potentially also be unified with dynamic loading. In general, there are many arguments in favour of bringing the worlds of operating system and language implementation closer together, for example the bad interactions between garbage collection and demand paging. Meanwhile, the arguments for the "revival of dynamic languages" [**?**] read like a manifesto for this closer integration, since most of the cited features—dynamic structural modification, persistence, namespaces, pluggable type-checking and reflection—are already, in some form or another, features of operating systems. I hope that my findings will add further support to these arguments, although it is unlikely that any substantial exploration will be feasible in the time available.)

5.3.3 Refactoring

The final piece of implementation is a refactoring tool, for human-assisted separation of integration from functionality within existing C code. Semi-automatic refactoring is rapidly maturing, and found in many popular development environments (notably Eclipse). Although most refactoring techniques are prototyped for Java or similar languages, refactoring C is also feasible with suitable treatment of the preprocessor. [**?**] The tool would accept single C source files, and (with user assistance) output two files: one, a simpler C source file implementing the "core functionality" in terms of *idealised* imports and exports; the other, expressed in the configuration language, detailing the adaptations necessary to recover compatibility of this simpler code with the original *non-idealised* imported and exported interfaces.

To intuit a possible algorithm for this refactoring, note that in any module of source code, there is a finite set of statements or expressions which perform communication with the outside (i.e. across some external interface). For each of these points, the algorithm may use the data dependency graph to search for logic which is candidate for shifting into the adaptation domain, to achieve the goal of better modularisation and/or lower overall complexity of the combined source and adaptation. Graph complexity measures, or even abstract syntax tree size, might be useful as heuristics for identifying such logic, but the semi-automatic approach allows fall-back onto human judgement.

6 Evaluation

I propose to evaluate the implementation work by a combination of two methods: case study, and software measurement.

6.1 Case study

Case study, as already described, will target some existing well-known open-source codebases, including Mozilla Firefox, and investigate the use of adaptation to produce novel combinations of code. I will target a selection of the dimensions described in Appendix A. Some example cases, and their dimensions, might be:

- adapting a web browser plug-in between different binary plug-in interfaces (e.g. Firefox to Konqueror);
- adapting a graphical application between different binary toolkit interfaces (e.g. GTK+ to Qt);
- porting a commonly useful feature from a web browser to a file manager (e.g. history or bookmarking);
- adapting a web browser extension to run as a stand-alone application in a separate process;

- adapting a graphical debugger front-end to use a new back-end with a differing command set or protocol;
- integrating a web-based on-line banking system with a home accounting program;
- any or all of the examples mentioned in the Motivation.

This "first pass" method of evaluation will proceed by demonstrating intuitively the simplicity of performing these tasks using the new configuration language and adaptation library, by comparison to conventional glue code.

6.2 Software measurement

A more rigorous assurance of success may be found by software measurement. For this purpose I will divide the proposed work into two: support for convenience of adaptation (i.e. the configuration language and its implementations), and support for refactoring (to effect the separation of functionality from integration in existing source code). Different measurements will be required for each.

In the case of the configuration language, we would like to show that using the language to perform adaptation and integration is cheaper than traditional methods (i.e. writing glue code in other languages). Direct user observation is possible, e.g. by giving coding assignments to a sample of undergraduates. However, it is difficult to factor out the familiarisation overheads associated with a new language. Instead, I propose to approximate "cheaper" with "less complex", and use code complexity measures. Many traditional complexity measures are unsuitable because they fail to account for relevant sources of complexity. For example, cyclomatic complexity considers only the control flow graph, so would not account for complexity inherent in a regular expression string-rewriting rule (which is a particularly complex string constant). Harrison's entropy-based measure [**?**] is an ordinal measure of *average information per token*, whose evaluation empirically demonstrates that this correlates negatively with bug density (a reasonable proxy for effective complexity). This work can likely be extended to allow comparison between my configuration language and traditional glue code, hopefully showing that the average information per symbol is greater in the new language and, correspondingly, that less total code entropy is required.

In the case of the refactoring engine, we wish to show that the refactored source is more composable, i.e. less *coupled*, than the original source. Existing measures of coupling (as first proposed by Stevens et al [**?**]) use ad-hoc weightings to assess the severity of coupling between each pair of modules. I propose a more principled measure, again based on information entropy—this is outlined in Appendix C. One complication is the fact that after refactoring there are at least three (rather than two) modules: the two being composed, and the adaptation or glue logic. As usual, the coupling of the ensemble can be measured as a weighted sum of the pairwise couplings of each component

with the adaptation logic, assuming that all communication proceeds through the adapter. (A trivial "identity function" adapter does not reduce coupling; it arguably increases it, since changes to one component might need to be reflected not only in the target component but also again in the adapter. The weighting must be chosen to reflect this.)

Evaluation will proceed by measuring the coupling of a refactored source tree, which makes use of adaptation features, and comparing it with the unrefactored version. Depending on the success of the refactoring engine, these refactored source trees might be produced semi-automatically using the tool, or else manually. Since the refactored interfaces should, intuitively, be simpler, and the adaptation logic also simpler than a traditionally written glue module, the measure should show a clear reduction in local coupling on both edges.

6.3 Non-criteria

One *non-criterion* regarding evaluation is performance. Clearly, the techniques described will necessitate greater indirection, greater numbers of procedures and module boundaries, run-time code generation, and other techniques which will degrade performance relative to a statically composed, statically optimised version of the same code. I am confident that the performance penalties can be substantially negated by relevant optimisation techniques. One existing example is the *flatten* technique employed by Knit [**?**]. However, I will not research such optimisations during this work.

7 Related work

Most relevant work is cited inline, including some in the Appendices. Here I highlight the recent work of most direct relevance.

Flexible Packaging DeLine's 1999 thesis [**?**] targets almost the same problem as this proposal: how can we compose functionality in the presence of mismatched integration details? However, Flexible Packaging takes a "clean slate" approach, rethinking the entire software development process. The result is a system which places strong constraints on the languages and styles in which code can be written, and cannot be applied to existing code. The approach proposed here, although less clean and offering less dramatic reductions in integration effort, embraces the wealth of existing code and permits greater heterogeneity.

Rich interfaces and adaptor synthesis There is considerable work on enriching component interface specifications with additional metadata [**? ?**], primarily for compatibility checking. Some of these interface specifications can be used as input to adaptor synthesis algorithms [**? ? ? ?**]. In most cases the capability of these algorithms is limited to finite-state protocol adaptations and argument permutations. This work is essentially

complementary to my proposal: it suggests some unifying notions of interface, and provides algorithms which might be incorporated into the library of adaptations.

Coordination Coordination languages such as Linda [**?**] and Reo [**?**] specify the interactions between concurrent computational processes, whether these be data flow (sends and receives) or control flow (waiting and resuming of processes). The non-invasive "exogenous" coordination [**?**] provides a separate configuration domain for expressing these interactions. This is consequently a domain convenient for performing adaptation of timing and protocol details. However, these coordination languages can't express changes to the individual messages sent and received, meaning that they can't express a large class of useful adaptations.

Linkage-level flexibility Some work has explored link- and load-time flexibility of binary code. Knit [**?**] introduces flexibility into the linkage graph for ahead-of-time linking. Load and Let Link [**?**] provides similar flexibility at run time. Binary Component Adaptation [**?**] relaxes composition constraints for Java bytecode by rewriting typing metadata. Like this proposal, all these works make the case for flexibility at the level of linking and loading. However, none of these addresses the problem of mismatched interfaces.

8 Future work and alternatives

For some ideas on future work and alternative avenues, see Appendix D.

9 Provisional structure and timetable

Below is a draft thesis outline, interleaved over a provisional timetable which begins in January 2008 (month 0) and ends with December 2009 (month 23). Please refer back to Sections 5 and 6 for details of all the implementation and evaluation work mentioned.

1. **Introduction**
2. **Technical background**
3. **Combining linkage with adaptation** I will describe the design and implementation of a linking language supporting an extensible set of adaptation primitives, as described in Section 5. I will also detail the experimental work applying the language implementation to a selection of case studies (including the "file manager history", "calendar sharing" and "debugger back-end" examples). From this experience, I will summarise and justify the language's underlying linkage model and a basic library of useful adaptation primitives.

- Months 0–3: begin work on Knit; extend to support multiple binary formats; add support for externally-defined adaptation primitives; implement argument remapping primitive as test
- 3–5: Knit-ify the necessary parts of two open-source codebases (provisionally Firefox and Rox Filer); identify and implement useful primitives for mix-and-match of features (e.g. history) between these
- 5–7: develop some useful case-study compositions, some simpler ones to include reference "old style" glue code implementations for later evaluation

4. **Dynamic composition with adaptation** I will describe the design and implementation of a variant of the linking language and its embedding into Linux's dynamic loader, as described in Section 5. I will describe and explain the deviations from the original language, and detail further case studies demonstrating dynamic composition use-cases (including the "browser natural-language translation plug-in" and "POP from file" examples, along with other mix-and-match of code targetting sockets, filesystem and database interfaces).

   - 7–9: embed into dynamic loading interface
   - 9–10: develop case studies for dynamic composition

5. **Refactoring** I will detail the implementation of a semi-automatic refactoring engine to separate functionality from integration in existing C code, as described in Section 5.3.3. I will report experiences of applying the engine to several of the case-study compositions already developed, showing that the refactored sources permit simpler adaptation logic.

   - 14–17: devise and implement refactoring
   - 17–18: apply refactoring to existing case-studies, using coupling measure to evaluate

6. **Entropy-based software measures** I will describe and justify the application of Harrison's entropy-based complexity measure to show that the *total information* and *average information per symbol* are both improved when using our adaptation techniques (as compared to traditional glue code). I will describe an entropy-based coupling measure which accounts for a broad range of sources of coupling. Although all this work logically follows the refactoring case studies, it is timetabled earlier, since limitations of the measure might constrain the choice of refactoring case studies.

   - 10–11: adapt entropy-based complexity measure to adaptation code, and evaluate old-versus-new (i.e. glue versus adaptation) using case studies already implemented

- 11–14: devise entropy-based coupling measure and test against existing measures

7. **Experimental evaluation and analysis** I will summarise the experience gained from the case studies made of the configuration language, showing both intuitive and measurable improvements, the latter using the complexity measure. I will also summarise experience from the refactoring case studies, again showing both intuitive and measurable improvements, the latter using the coupling measure.

8. **Conclusions and future work**

   - 18–20: leftover or additional experiments and practical work
   - 20–23: final-phase dissertation writing
   - 23: submission

References

[] R Allen and D Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6:213–249, 1997.

[] C Andreae, J Noble, S Markstrum, and T Millstein. A framework for implementing pluggable type systems. *ACM SIGPLAN Notices*, 41:57–74, 2006.

[] F Arbab. What do you mean, coordination. *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, 1122, 1998.

[] F Arbab and F Mavaddat. Coordination through channel composition. *Coordination Languages and Models: Proc. Coordination*, 2315:21–38, 2002.

[] DM Beazley. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.

[] AD Birrell and BJ Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2:39–59, 1984.

[] A Bracciali, A Brogi, and C Canal. A formal approach to component adaptation. *The Journal of Systems & Software*, 74:45–54, 2005.

[] G Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[] N Carriero and D Gelernter. Linda in context. *Communications of the ACM*, 32:444–458, 1989.

[] A Chakrabarti, L de Alfaro, TA Henzinger, M Jurdzinski, and FYC Mang. Interface compatibility checking for software modules. In *Proc. 14th CAV, LNCS*, volume 2404, pages 428–441.

[] EM Dashofy, A van der Hoek, and RN Taylor. A highly-extensible, xml-based architecture description language. *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103–112, 2001.

[] L de Alfaro and TA Henzinger. Interface automata. *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, 2001.

[] R DeLine. *Resolving Packaging Mismatch*. Phd thesis, Carnegie Mellon University, 1999.

[] E Eide, A Reid, J Regehr, and J Lepreau. Static and dynamic structure in design patterns. *Proceedings of the 24th international conference on Software engineering*, pages 208–218, 2002.

[] T Ewald. Overview of com+. In G Heineman and WT Councill, editors, *Component-based software engineering: putting the pieces together*, pages 573–588. Addison Wesley, 2001.

[] D Garlan, R Allen, and J Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *Proceedings of the 17th international conference on Software engineering*, pages 179–185, 1995.

[] C Haack, B Howard, A Stoughton, and JB Wells. Fully automatic adaptation of software components based on semantic specifications. *Algebraic Methodology & Softw. Tech., 9th Int?l Conf., AMAST*, 2002.

[] Warren Harrison. An entropy-based measure of software complexity. *IEEE Trans. Softw. Eng*, 18:1025–1029, 1992.

[] TA Henzinger. Rich interfaces for software modules. *Proc. of the 18th European Conf. on Object-Oriented Programming (ECOOP 2004). Berlin: Springer-Verlag*, pages 516–517, 2004.

[] SP Jones, E Meijer, and D Leijen. Scripting com components in haskell. In *Proceedings of ICSR5*, 1998.

[] R Keller and U Holzle. Binary component adaptation. *ECOOP*, 98:307–329, 1998.

[] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th European Conference in Object Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[] J Magee, N Dulay, S Eisenbach, and J Kramer. Specifying distributed software architectures. *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, 1995.

[] B McCloskey and E Brewer. Astec: a new approach to refactoring c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 21–30, 2005.

[] J Misra and WR Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, pages 83–110, 2006.

[] J Mukherjee and S Varadarajan. Develop once deploy anywhere: achieving adaptivity with a runtime linker/loader framework. In *Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, pages 1–6, 2005.

[] O Nierstrasz and F Achermann. Separation of concerns through unifica-

tion of concepts. *ECOOP 2000 Workshop on Aspects & Dimensions of Concerns*, 2000.

- O Nierstrasz, A Bergel, M Denker, S Ducasse, M Galli, and R Wuyts. On the revival of dynamic languages. *Software Composition: 4th International Workshop, SC 2005, Edinburgh, UK, April 9, 2005: Revised Selected Papers*, 2005.
- R Passerone, L de Alfaro, TA Henzinger, and AL Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the International Conference on Computer-Aided Design*, 2002.
- JM Purtilo and JM Atlee. Module reuse by interface adaptation. *Software - Practice and Experience*, 21:539–556, 1991.
- Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, 2000.
- J Ren, R Taylor, P Dourish, and D Redmiles. Towards an architectural treatment of software security: a connector-centric approach. *ACM SIGSOFT Software Engineering Notes*, 30:1–7, 2005.
- M Shaw. Architectural issues in software reuse: It?s not just the functionality, it?s the packaging. *Proc IEEE Symosium on Software Reusability*, 1995.
- M Shaw, R DeLine, DV Klein, TL Ross, DM Young, and G Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21:314–335, 1995.
- WP Stevens, GJ Myers, and LL Constantine. Structured design. *IBM Journal of Research and Development*, 13:115, 1974.
- DM Yellin and RE Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19:292–333, 1997.

Appendices

A Dimensions of adaptation

When discussing adaptation, it is helpful to describe the concrete ways in which two components may be mismatched. Here I will enumerate some non-orthogonal dimensions of adaptation.

Data encoding Data of the same *meaning* may be concretely represented in many different forms. For example, there frequently exist many different file formats, character sets or network protocol messages for what are, at some higher level, the same meanings. The simplest cases of such mismatch may be handled by conversion routines, translation tables or

other mapping constructs. I discuss more specific cases of this mismatch in the following paragraphs.

Operations Units of code may implement logically compatible operations but differ in the concrete expression of their interfaces. For example, two traditional procedural or object-oriented interfaces might differ in the names of operations, order and types of arguments, and type of return value. This is a special case of the data encoding mismatch, since arguments and return values may be thought of more generally as structured messages. These mismatches might occur at a higher semantic level than that of conventional type systems – for example, a procedure

```
substring ::  string → int → int → string
```

might interpret the two integers either as `(start, end)` or as `(offset, length)`.

Protocol A complement of the syntactic operational mismatch is the more semantic issue of protocol mismatch. In any stateful communication, the meaning of any message depends on what messages have been sent previously, and in what order. These messages might be those of network protocols (an obvious candidate for mismatch, especially versioning issues), procedural interfaces (where each call is a pair of messages) or even variable accesses and initialisations (for example in C, where the semantics of global and local variables are subtly different with respect to initialisation protocol). Finite-state automata may be used to capture many protocols [**?** **?**], although some may require more complex automata (e.g. a stack which may not be popped more times than it has been pushed, requiring a deterministic pushdown automaton [**?**]).

Language A common difficulty is combining code written in different languages. First, there must be some *model* of the relevant dynamic constructs of one language expressed using those of the other. Ensuing subtleties include expressivity (that all meaningful operations provided by the foreign-language component can invoked on its native model), safety (that no undefined operations can be invoked through the model), and efficient implementability. Procedure call and other message-based communication is easily handled by value translation (i.e. marshalling), and tools exist to generate translating wrappers (e.g. Swig [**?**]). Some language features require new implementation techniques (e.g. spaghetti stacks) which are less straightforward to integrate with other runtimes. Sharing state is also complex, since state management semantics vary greatly (e.g. automatic versus manual) and have far-reaching implementation consequences. A final difficulty is from "closed world" assumptions which break when resources are shared: consider a garbage collector which stops all other threads. Finding *all* other threads is easy under the assumption that these are only those created by a unique runtime library. The ability to interpose *within* the language implementation, to adapt this logic, is

therefore essential for supporting integration. Nevertheless, interoperation between *specific implementations* is essentially similar to any other problem of mismatched library composition.

Targeted API When programs are written against pre-existing concrete interfaces, such as system call interfaces or library interfaces, they are inevitably coupled to these interfaces. It often proves desirable, but expensive, to port software to target a different piece of supporting software – perhaps a different operating system, a different windowing toolkit, a different mathematical library, and so on. A convenient way of generating a suitable wrapper, with little or no change required to the existing code, would be a very useful form of adaptation.

Binary interface Essentially a special case of data encoding mismatch, ABI mismatches are caused by different conventions for data representation and communication. In compilers they are worked around by building in knowledge of several different ABIs, and by annotating source code (for example using `extern "<lang>"` directives in C). This works satisfactorily only because of the small number of conventions commonly implemented for any given instruction set architecture. More generally there needs to be a systematic way of specifying ABIs and performing the necessary interposition or rewriting.

Programming style For any given functionality, there may be many programming styles, above the level of language, which may be used to implement it. Classic examples are the event- versus thread-based styles, the filter versus the in-place update, data-flow versus control-flow, and so on. A particularly powerful form of adaptation would be capable of mediating between logically compatible code written in different styles, enabling composition of highly heterogeneous codebases.

"Packaging" Although there exist innumerably many possible conventions for binary interface, programming style and the like, in practice we observe only a very small subset of these possibilities. For example, when writing some new code, we would choose to "package" it in one of several different ways: as a command-line tool, a graphical tool, a C library, a Python library, a web form, a spreadsheet macro, or various others [**?**]. Crucially, we would clearly pick *some* pre-existing set of communication conventions, rather than inventing our own. A particularly pragmatic approach, therefore, is to develop means of adapting between pre-existing (and future) packagings, for some one-time effort in each case, without going so far as to achieve the level of generality required to support any conceivable packaging with equal effort.

Software architecture Code frequently makes assumptions about communication topologies, control structures, causality of information flow, and reasoning about uniqueness or completeness conditions over the remainder of the system. Among the hardest and most subtle kinds of mismatch

are those arising from such assumptions. Garlan et al [**?**] provided a case-study detailing several such problems: inability to decompose and minimise run-time code dependencies, inability to extend event loops, inaccessibility of desired interfaces to objects, introduction of unwanted multithreading, and overly constrictive concurrency control. The underlying assumptions causing these problems are usually not stated explicitly in source code, and can therefore be difficult to identify. Adaptating the components to overcome these mismatches – by extending event loops, decomposing dependencies, exposing internal interfaces and altering concurrency control logic – is a challenging task.

B Survey of existing practices

Scripting languages Scripting languages are programming languages characterised by brevity, support for dynamic code evaluation, and lack of static type-checking. They exploit a trade-off: in return for a loss of some efficiency, elegance and safety, they gain dynamism, convenience and ease of modification. Scripting languages are therefore popular for "glue code", whose purpose is to interface existing pieces of software. Glue code invariably performs adaptation, and has special support for this in the extensive regular expression-based string matching and rewriting found in mainstream scripting languages such as Perl and Python. Additionally, these languages provide convenient support for interacting with external code using a wide variety of system-level communication mechanisms: invoking other scripts, accessing the file system or network, invoking external programs, manipulating environment variables, and so on. However, lack of static checking can make script less reliable than code in conventional languages. Also, adaptation by string rewriting is especially error-prone—but is necessary because of the low-level byte-stream IPC mechanisms by which scripts and other components traditionally communicate.

Orchestration and service-oriented computing Related to scripting is the recent trend towards "service-oriented computing", where large systems are decomposed into a set of passive services (typically web services or other RPC-like abstractions) and a set of proactive control components which use them. These control components might be called as orchestrations, workflows, or simply scripts. Separating the proactive from the passive allows new languages to be used to express the active components, which may have convenient support for error-handling, parallelism, and asynchronous or high-latency communication [**?**]. This support has proved useful for building distributed applications in the wide area, in addition to providing the benefits of scripting languages. However, as with scripts, there may be considerable effort in adapting between the combination of multiple proactive components.

Distibuted middleware Middlewares frequently use IDL compilers to gener-

ate communicational code, supported by run-time libraries [**?**]. This frees application code from the need to build in particular implementations of communication abstractions (such as remote procedure call). However, since client and server code must share a common interface, and must be written to the conventions demanded by the middleware's communication abstraction (and perhaps also to the syntactic conventions of the particular IDL compiler being used), this often does not help in the case of combining code written independently,

Component middleware So-called "component-based" technologies such as JavaBeans, COM+ and the CORBA Component Model have become popular in industry for creating components of richer interface description (and hence greater perceived re-usability), for composing such components (often graphically), and for automatically generating certain kinds of marshalling wrappers (such as COM+'s context proxies [**?**]). Note that marshalling is a form of adaptation, at the level of data representation. While useful, these technologies do not support any higher-level forms of adaptation, such as adapting between mismatched interface definitions. Therefore, although their emphasis on interface specification is helpful, they do not solve the problem of direct re-use of independently developed code.

Programming language advances Higher-level programming languages, including functional and higher-order languages such as Haskell or ML, provide a more powerful set of basic abstractions than many traditional languages. These include tuples, lists, streams, first-class functions, discriminated unions and pattern-matching. The inclusion of such abstractions cuts down the potential for mismatch which might otherwise be caused by differing conventions or implementations of these common abstractions. Meanwhile, the powerful computational abstractions of lazy evaluation and higher-order functions might enable more convenient expression of script-style adaptation logic [**?**]. However, the need to adopt a common language, and the longstanding inconvenience of interfacing these languages with foreign code, mean that for the foreseeable future there will be greater need to perform adaptation *to* and *from* these languages rather than *within* them.

Configuration languages "Configuration languages" include roughly any language which expresses wiring, linkage, component topologies, component initialisation data or other specialisations particular to a specific deployment. Examples are linking languages [**?**], architecture description languages [**? ? ?**], exogenous coordination languages [**?**], the configuration languages of "inversion of control" development platforms (such as Spring[2] or Castle[3]), and, strictly speaking, almost all conventional programming

[2] `http://www.springframework.org/`
[3] `http://www.castleproject.org/`

languages. (We informally exclude the latter, for convenience of reference to the remainder.) These languages are useful for conveniently separating concerns, enabling static checking [**?**] and resolving some lower-level mismatches (e.g. of component naming or wiring). However, existing configuration languages do not support any higher-level forms of adaptation. We will return to this idea in Section 4.

Aspect-oriented programming This technique [**?**] extends programming languages with a new kind of module called an *aspect*, which specifies inline insertions or modifications to code within other modules, at certain "join points" specified declaratively by the aspect definition. An aspect can be used to modularise features whose code might otherwise be scattered throughout many modules—such features might include logging, security checks, concurrency control and so on. Aspects can be treated as first-class units of composition (i.e. linkage) alongside traditional modules, and it may be convenient to effect certain adaptations either as new aspects or as changes to existing aspects, instead of making inline changes to a large set of modules. As such, aspect weaving can be seen as a particularly general adaptation primitive. However, the technique has not yet been specialised towards composition of heterogeneous multi-language systems—for example, most aspect toolchains target a particular language, while inter-module linkage is typically specified endogenously by name-matching rather than in a separate configuration domain.

Unification of programming interfaces The Unix motto of "everything is a file" is an instance of a general technique: defining a unified programming interface onto a disparate set of objects. The intention is to maximise the composability of application code with respect both to data, such as files, and to other pieces of application code—where communication with this code is itself abstracted by the unified interface (as with pipes). Other examples are the BSD sockets API, and the World-Wide Web with its small set of HTTP "methods". This approach is appealing, but has drawbacks. Unification comes at the expense of semantic detail, so little static checking can be performed. In practice, most objects implement some ill-defined subset of the unified interface, discoverable only at run-time by query or, worse, only in error-handling. Worst of all, some operations of some objects simply will not be mappable satisfactorily from the unified interface; this forces either an arbitrary local choice among the many unsatisfactory ways, or the use of an escape-hatch such as Unix's `ioctl()`. In both cases, the original benefit is lost, since there is now a high likelihood of mismatch with other application code. Adding a layer of indirection (i.e. adaptation) between these pieces of application code is a promising solution.

Unification of binary interfaces Similar to API unification, unifying binary representations and linkage models brings immediate interoperability benefits. Testament to this fact is the successful implementation of a large

number of languages over Microsoft .NET's Common Language Runtime. Since byte-code is almost always generated by tools rather than by hand, code changes are not a problem: it is sufficient to implement a compiler from each source-level language to the common byte-code. However, again the need for semantic uniformity can be restrictive: in the .NET case, all languages must use a garbage-collected heap, the .NET threading model, a common data model, a common type system, and so on. Moreover, no standard is ever final, nor adopted everywhere, so there will always be a need for adaptation between different binary-level conventions. This is evidenced by the current market for Java-to-.NET interoperability products, including bytecode translators[4] and trampoline generators[5].

Metadata and annotations Recent languages and linkage standards, including both Java bytecode and .NET intermediate code, incorporate the ability to annotate sections of code with arbitrary metadata. This is useful for making explicit the semantic distinctions between apparently unified objects, such as methods, variables, and so on. However, clearly it is also necessary for application code to take these annotations into account. Therefore, annotations are a useful set of inputs into the adaptation process, but do not in themselves solve the problems of adaptation.

C Proposed entropy-based coupling measurement

Existing software measurement work defines various measures of *coupling*, including some specific examples measuring *local coupling*. This concept was introduced by Stevens et al [**?**] as "the measure of the strength of association established by a connection from one module to another". One consequent intuition is that coupling measures the likelihood that changes in one subsystem will require consequent changes in a disjoint subsystem, and therefore low coupling predicts good properties such as extensibility and maintainability. Another intuition is that high coupling correlates negatively with reusability, since the more strongly a module is coupled with its environment, the more changes are necessary in order to re-use that module in a different environment.

We would therefore like to show that compositions making appropriate use of adaptation are less strongly coupled than traditional compositions. Unfortunately, existing measurements for local coupling are ad-hoc and do not capture all sources of coupling. I propose a new measure based on information theory and a channel-based model of communication.

To illustrate, consider an untyped, unstructured communication interface such as Unix pipes or files. In order to communicate structured data, the parties must fix on conventions for coding that structure, typically a combination of whitespace and punctuation characters. The convention chosen must be understood by both parties in order for them to communicate, and is therefore

[4] IKVM.NET: `http://www.ikvm.net/`

[5] JuggerNET: `http://codemesh.com/products/juggernet/`

a source of coupling. Now consider a further intuition: if a component is able to understand multiple alternative structure codings—for example, if it detects whether commas or tabs are being used to delimit input fields, and interprets the input correctly in either case—then it is less coupled to its environment than if it understands only one. This is because the component and the environment have to agree on *fewer choices* about the code in the former case than in the latter. This *number of choices* idea clearly suggests information entropy, and it is specifically the entropy of the channel's *coding rules* which correlates with coupling.

I propose that the coupling between two modules joined by adaptation can be measured by the *complexity* (i.e. entropy) of the shared interface definitions on which the modules depend. Suitably advanced notions of interface are required for this to capture all meaningful sources of coupling, certainly beyond the highly syntactic interface definitions used in today's code. For example, in the Unix pipeline

```
printenv | sed 's/=.*//'
```

which prints out the names of all defined environment variables, there is an implied contract that all lines output by `printenv` follow the pattern `NAME=value`. Temporal, timing- and protocol-based interfaces [**?**] are still useful for describing these contracts, but must be applied with sufficiently fine grain to capture the complete *language* of individual data values sent and received, looking further just the language-defined constructs such as procedure signatures. As an alternative to these, I may explore a new, purely channel-oriented notion of interface, based on the concepts of *symbols*, *symbol content* (i.e. the symbol itself), *symbol context* and *context variables*.

D Future work and alternatives

Pluggable checking So far we have considered only the problem of creating working compositions of software. A separate problem is the ability to reason about these compositions and check arbitrary properties of them (including type-safety, but potentially also quality of service, security and so on). Given our emphasis on dynamism and heterogeneity, a useful approach is that of *pluggable checking* [**?** **?**], where type systems and other reasoning frameworks are composable extras applied to configurations of software. It would be interesting to add support for annotations, both as discovered properties of code and as external assertions within the configuration language, and demonstrate some examples of pluggable checkability over these (and over intervening adaptation primitives).

Top-to-bottom traceability Configuration languages span a spectrum from high- to low-level. At the high level, architecture description languages (ADLs) such as Unicon [**?**], Wright [**?**] or xADL [**?**] describe the structure of large, possibly distributed applications. With the advent of

virtualization technologies such as Xen and VMware, system software now also supports the description of entire distributed applications spanning multiple machines in the local area. (Specifically, in the case of Xen, these descriptions would be the inputs into a domain builder for distributed applications.) This opens up the possibility of describing the intended structural and extra-functional properties, using ADLs, and having the system toolchain and runtime directly support the checking, enforcement and traceability of those properties. These properties might also usefully include security properties and policies [**?**].

Adaptation-oriented programming If all programmers had access to convenient and expressive features for adaptation, how would this change the way programs are written? Intuitively, the need to target concrete pre-existing interfaces (such as library interfaces) when writing new code seems to cause a "leakage" of complexity—where the highly general-purpose library interface imposes unwanted complexity onto what might otherwise be a much simpler client. In other words, targeting existing APIs increases the complexity of the resultant source code, and reduces its reusability. Writing code which deliberately ignores pre-existing targetable interfaces, with the expectation of using adaptation to perform this integration later, might reduce client complexity and hence improve re-usability of this new code. Investigation of this phenomenon might prove a useful alternative avenue.