A Survey of Practical Software Adaptation Techniques

Stephen Kell
(Computer Laboratory,
University of Cambridge, United Kingdom
Stephen.Kell@cl.cam.ac.uk)

Abstract: Software adaptation techniques appear in many disparate areas of research literature, and under many guises. This paper enables a clear and uniform understanding of the related research, in three ways. Firstly, it surveys a broad range of relevant research, describing and contrasting the approaches of each using a uniform terminological and conceptual vocabulary. Secondly, it identifies and discusses three commonly advocated principles within this work: component models, first-class connection and loose coupling. Thirdly, it identifies and compares the various modularisation strategies employed by the surveyed work.

Key Words: adaptation, coupling, reuse, communication, coordination, software composition, modularity, software architecture, software measurement
Category: D.2.6, D.2.7, D.2.11, D.2.12, D.2.13

1 Introduction

Like all systems, software systems consist of multiple interacting parts. As these systems become more complex, and the space of applications becomes ever larger, it becomes more economical to develop them by *composition*, *extension* and more generally *re-use* rather than from scratch. These goals, in turn, demand understanding of the individual system parts: how they communicate dynamically in a running system, and how we describe them statically as executable code.

Widespread re-use of software poses two specific difficulties: designing a system in pieces such that each might individually be re-used (which I will call *modularisation*), and composing a required system out of whatever pieces are already available for re-use (which I will call *mismatch*). The latter is a common problem, since units of software developed independently are unlikely to be directly compatible. Rather, some additional work is necessary to join them together appropriately, overcoming the mismatch—this is adaptation. Meanwhile, the desire to specify adaptation in a different style, notation or language from other code means that much adaptation work also proposes a novel approach, or novel criteria, for modularisation.

There has been much research aimed at tackling these problems, within many communities and under a large number of headings. These headings include software composition, adaptation, coordination, software architecture, module systems, interconnection languages, linking languages, and more. Unfortunately, the large number of sub-areas into which this related work is divided makes it

difficult to acquire a uniform understanding of the whole. I hope to facilitate better understanding through the following contributions.

- I survey several research fields related to adaptation, describing and contrasting their approaches using a consistent conceptual and terminological vocabulary.
- I identify three broad themes advocated by much of the existing surveyed work: explicit connection, component models and loose coupling. For each, I summarise and discuss the various arguments found in the literature, exposing subtle differences in motivation, application and terminology.
- I identify the class of *configuration languages* running through the literature, discussing several different modularisation strategies advocated by the surveyed work and showing how they overlap.

I begin by surveying several existing research areas and their contribution to adaptation.

2 Guises of adaptation

One difficulty faced by adaptation researchers is that relevant work is spread over many fields and communities. Much relevant work does not describe itself as "adaptation". In this section, I make a non-exhaustive survey across several research fields, using a uniform vocabulary to describe and contrast the work in each. My emphasis will be on existing *practical* work relating directly to adaptation.

2.1 Definitions and exclusions

I define adaptation as any process which modifies or extends the implementation or behaviour of a subsystem to enable or improve its interactions, or synonymously, its *communication* with the surrounding parts of the system (which we call its *environment*). Note that "communication" here includes not only dynamic interactions at run-time, but also interactions occurring statically, perhaps in the compiler [see 1].

From this definition, we may distinguish multiple kinds of adaptation. The first is adaptation done for *functionality* and *correctness*—the ability for two subsystems to exchange information in a way which preserves *meaning*. This is the kind of adaptation which relates most fundamentally to re-use.

[1] This may seem strange, but consider that we are concerned only with the input artifacts to the composition process, and not whether our tools happen to precompute certain interactions—which is highly dependent on implementation. I simply state this for now, but will return to this interpretation of communication in [Section 3.3].

A second kind is adaptation for the sake of extrafunctional properties, such as performance, reliability, security, quality of service and so on. This is a much harder problem, in that components lacking these innate qualities are more likely to require deeper changes. However, I will survey some especially invasive and low-level techniques that are capable of performing such adaptations.

In a closed system, "adaptation" or "adaptive" can refer to the ability of a system to configure itself in response to changing network conditions, resource constraints and the like. This is a form of adaptation for the sake of *optimisation*. Since they do not involve novel compositions of code, but rather run-time selection among a closed set of modules, I will refer to this kind of adaptation as "dynamic reconfiguration", and subsequently use "adaptation" to refer only to the other kinds. I do not cover any work on dynamic reconfiguration in this paper.

Finally, we note that in an open system with a dynamically changing component population, such as a mobile environment, a particular adaptation problem arises: as devices encounter other devices, it is desirable to *automatically* and *dynamically* construct an adaptor which will allow them to successfully communicate. Several of the synthesis techniques surveyed may serve as a basis for this; however, they must be augmented by some further automatic techniques, such as ontology matching, in order to bootstrap meaningful communication. These latter techniques are outside the scope of this survey, which covers user- or developer-guided techniques only.

2.2 Taxonomy

There are several dimensions along which different adaptation techniques may be clearly distinguished. In order to compare and contrast adaptation techniques, I introduce a few simple taxonomies over the various properties of adaptation techniques.

Input representations What representations of software are targeted by the technique? This is typically either source code, perhaps restricted to particular languages, or else binaries of particular formats. Some techniques require multiple kinds of input, each in a language with some particular role or expressive bias. I therefore split language roles into the following *notation classes*: *implementation*, the typical role of conventional programming languages; *specification*, meaning high-level domain-specific languages for describing some intent or constraint (typically used as input to synthesis algorithms); and *configuration*, description of the structure and nature of relationships among *multiple* components. Examples of configuration languages include all architecture description languages and linking languages; strictly speaking, almost all implementation languages may be included, but

I only use "configuration languages" to refer to those which are *not* designed for use also as implementation languages. Finally, certain metadata may also be part of a technique's input notation, particularly when targeting binary representations—this is described in the next paragraph.

Domains of adaptation Some techniques let us arbitrarily alter the implementation of existing components. Most, however, are constrained to one or more particular domains, which constrain the kinds of mismatches that it can resolve. For example, a technique might let us make changes or additions only around the interfaces to components. I distinguish four domains of adaptation: *implementation*, the most general, which allows arbitrary changes to arbitrary pieces of implementation; *structure*, which concerns changes only to the wiring or referential topology of the system; *messages*, which concerns the rewriting or replacement of data items sent between communicating components (whether these data items be procedure invocations, database queries, packets in a network or files on disk, etc.); *timing* or *sequencing* which concerns the ordering and timing constraints over messages; and *metadata*, which refers to descriptive elements such as typing metadata of components. (The latter do not directly concern the functionality of a component, but may need to be adapted to enable composition in the presence of type-checkers, signature verifiers or other extra-functional checks.)

Modes of adaptation Given our split of input representations, we can also describe *what kind of process* yields or achieves the adaptation, and usually *how adaptations are represented*. The process of adaptation is always either an *edit*, meaning an invasive change to one or more of the input notations, or else involves some separate entity which we call an *adaptor*. Adaptors may be introduced by *definition* from scratch in some notation (which may or may not be one of the input notations), by *selection* from a set of known adaptors, or by *synthesis* from a description in some specification notation. Adaptors themselves are always classifiable as either *deltas* or *containers* with respect to one or more input components. The distinction here is that containers are opaque and contain complete specifications of their own interfaces, as in the adaptor pattern (see [Section 2.3]), whereas deltas are transparent and expose the underlying original components' interfaces wherever these are not explicitly modified, just as with implementation inheritance or mixins [Bracha and Cook 1990] in OO languages. For deltas and containers it is also useful to distinguish whether it can use knowledge of the entirety of the input notation, in which case it is called *white-box*, or only on some interface-like abstraction of the input, when it is *black-box* [see 2]. Deltas

[2] I also use the terms *invasive* and *non-invasive* synonymously with white-box and black-box respectively. Note that I do not distinguish whether or not the particular *im-*

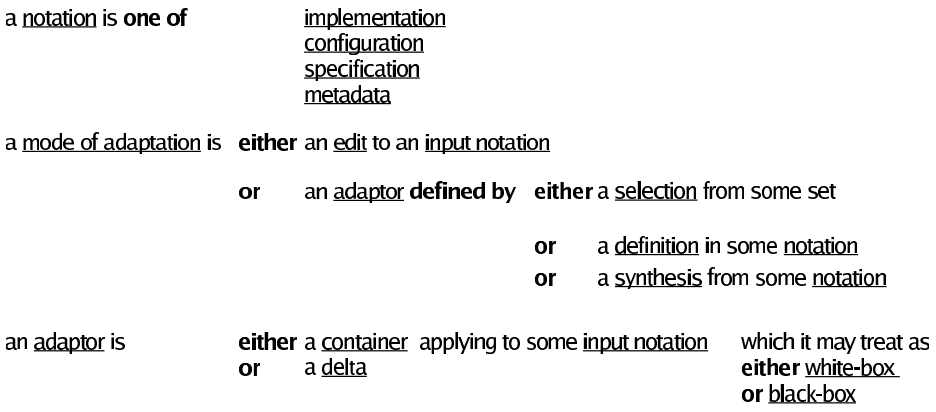

Figure 1: The space of modes of adaptation

and containers often target exactly one component, but may target more. An *intermediary* is a black-box delta which joins two or more components. A *wrapper* is a black-box container applying to one or more components. [Fig. 1] summarises the space of these modes of adaptation.

It is also useful to illustrate these taxonomies with concrete examples of what the technique supports. For this reason, I will also highlight what concrete *representation of adaptation logic* a technique supports (a refinement of its domains of adaptation), and what concrete *range of primitive adaptations* is provided (a refinement of the modes of adaptation).

Finally, some derived properties are of interest. Most adaptation techniques are a *manual* process, but we consider those involving synthesis to be partly *automatic*. Note that even these techniques rely on some kind of initial (hand-written) specification. Adaptations can be supplied at various times: perhaps *ahead-of-time*, at *compile time* at *load time*, or at *run time*. The adaptation might be applied straight away, meaning *eagerly*, or might be delayed for a while for optimisation reasons. Finally, we have already mentioned *invasiveness*, but must be careful to distinguish three separate issues: *edits*, which are adaptations made by changing input notation; adaptors *described invasively*, i.e. in terms of the internal details within an input notation, but modularised separately; and adaptors *implemented invasively*, perhaps for efficiency, where these may or may not be defined with reference to internal component details (consider

plementation of a delta or container examines the internals of the target components—only whether their definition may logically *depend upon* those internals. Efficiency concerns, for example, might motivate the use of invasive "white-box" program transformation techniques to implement what is logically a black-box container.

using a load-time optimiser after performing otherwise non-invasive black-box adaptation).

I begin with brief summaries of relevant conventional programming practice, and scripting techniques, continuing into work which describes itself using the term "adaptation", and then beyond into other relevant fields. Throughout, I have grouped work according to the words and terminology by which that work describes itself, since this correlates reasonably with the research communities from which each piece of work originates, while avoiding the inevitable subjectivity of attempting to judge conceptual similarity. Where particularly clear conceptual similarities exist, I provide cross-references. In addition to the text, a summary table is provided at the end of the survey.

2.3 Conventional programming practice

The simplest and best-understood way to adapt a piece of software is to copy its source code and make the necessary changes. This practice is commonplace, and for some kinds of software, notably device drivers, is often anecdotally reported as the dominant approach. Its disadvantages are many and obvious: it is labour-intensive, error-prone, requires a high degree of program comprehension by the adapting programmer, is only applicable when source code is available, and yields a forked and non-reusable adapted component which must be redeployed in its entirety. This unsatisfactory technique is the baseline upon which all techniques presented in this survey are trying to improve.

Adaptation is familiar to programmers in conventional object-oriented languages thanks to the "adaptor design pattern" described by [Gamma et al. 1995]. This itself was proposed as a more flexible alternative to implementation inheritance: while the latter effectively defines a *delta* over the overridable methods of a statically chosen superclass definition, the former provides a *wrapper* over a dynamically chosen target object. The "adaptor pattern" is simply the documented practice where an object with one interface is rendered composable with clients of a different second interface, by interposition of an adaptor object which consumes the first interface and exposes the second. It is, of course, a useful programming pattern which has been employed for decades, but in this survey we are interested in techniques for specifying and performing adaptation *more effectively* than these conventional approaches.

It is also worth noting that in conventional software development terminology, "porting" (e.g. of programs to new run-time environments, or features to new codebases) is synonymous with our definition of adaptation. It is typically performed by one or both of the above two techniques: wholesale edit-based modification of all environment-specific code, or creation of a "compatibility layer" adaptor.

Adaptors also appear in the world of distributed middleware as a general-purpose point of interposition. CORBA, a distributed object middleware standard, features both a "basic object adaptor" and later a "portable object adaptor". These are server-side objects responsible for dispatching incoming requests to other server-side objects, which may or may not implement the interface being consumed by the client [see Pilhofer 1999]. Again, this provides a highly general interposition mechanism but does not contribute any special techniques for performing adaptation.

2.4 Scripting and glue

There is a common distinction in programming practice between "scripting" languages and other "systems" or "component" programming languages. Although expounded in comparatively recent literature, notably by [Ousterhout 1998], the distinction goes back far into history: at least to the 1970s and the Unix shell, and perhaps further still. Certain attributes are usually associated with scripting languages: lack of static type-checking, brevity, expressivity, and support for dynamic code evaluation. The key underlying characteristic is the trade-off of safety and performance for brevity and dynamism [see 3]. The combination of dynamism and brevity makes it quicker to alter scripts—i.e. to perform *edit-based* source-level adaptation on script components—than to modify components written in conventional languages.

The composition language Piccola [see Achermann and Nierstrasz 2001] is founded on this latter observation: its mantra "applications = components + scripts" intends that scripts should be used for parts which are less stable, since they are likely to be more convenient to modify invasively [see Nierstrasz and Achermann 2000]. Piccola is a scripting language with formal semantics based on an extended pi-calculus. It allows the definition of customised *compositional styles*, each with their own rules and composition operators. These operators may be tailored to particular domains, such as GUI component composition, Unix-style pipelines, or various object-oriented composition techniques. Although adaptors might be conveniently integrated into a compositional style as special composition operators, Piccola provides no special support for implementing adaptors themselves.

As well as being more amenable to edit-based adaptation, scripting languages often have special features for adapting the components they script. Mainstream

[3] Ousterhout claims that typed interfaces "prevent other types of objects from being used ... even where that would be useful". Perhaps so, but a better solution is to improve type systems rather than abandon them. Brevity of scripting languages comes in part from eliminating type annotations; a better trade-off might be to add type inference. The "uniform representation" offered by text-based data interchange is a serious *hindrance* to re-use, since different developers may devise countless incompatible textual encodings for their data, entailing the use of error-prone techniques such as regular expression-based rewriting.

scripting languages such as Perl and Python have extensive support for regular expression-based string matching and rewriting. This is useful for adapting data between different encoding conventions; it is also error-prone. These languages also support invocation of external code using a wide variety of communication mechanisms: calling other scripts, accessing the file system or network, invoking external programs, manipulating environment variables, and so on. This "Swiss army knife" philosophy makes them convenient for glue code which integrates components that are *heterogeneous* with respect to their chosen communication mechanisms.

2.5 Adaptation by name

Adaptation of procedural interfaces was first directly addressed by the Nimble system [see Purtilo and Atlee 1991]. It provides a new pattern-based language for rewriting the arguments and return values of procedure call. It can reorder, duplicate or omit arguments, or supplement them based on default values, and can also call on external functions where necessary to convert representations and types. However, it lacks the statefulness necessary to express adaptations over *sequences* of invocations.

[Yellin and Strom 1997] address this shortfall in their work on finite-state protocol adaptation. This work brings a semi-automatic approach in which each component's source-code is annotated with a finite-state protocol description, and a complete adaptor is synthesised from a high-level partial specification. [Passerone et al. 2002] add a game-theoretic interpretation of the adaptor synthesis algorithm, showing that the process of synthesis is equivalent to that of testing for synthesisability. [Bracciali et al. 2005] replace finite-state automata with a replication-free pi calculus. This remains finite-state, but introduces the *channel* abstraction, hence capturing the dynamic evolution of communication structure.

Binary component adaptation for Java (BCA) [see Keller and Holzle 1998] implements a load-time adaptation for Java binaries. Adaptations to a class definition, such as member renamings and method additions, are specified in a "delta file" using a special language syntactically resembling Java. However, the range of adaptations available is narrow, being essentially limited to addition of new code, renaming of symbols, and changes to typing metadata (such as effectively adding an "implements" clause). Later work under the heading of "program transformation" (see [Section 2.8]) extends these abilities.

AxML [see Haack et al. 2002] supports adaptation over a purely functional subset of Standard ML. Programmers annotate candidate adaptation code with specification axioms, built from arbitrary predicates whose meaning is opaque to the tool. The axioms are expressed in propositional logic extended with universal quantification. Adaptation is triggered by defining data structures or func-

tors as "synthesis requests" which describe, using the same vocabulary of predicates, the signature type and specification of the required code. The adaptation is performed by source transformation, at compile time. Since it operates on purely functional code, the domain of AxML is effectively the same message-based transformations as with Nimble, but with added support for higher-order features (such as adaptation between curried and uncurried versions of the same function). This range of primitives suffices because the target code contains no explicit state or communication, unlike procedural code.

The work of [Rine et al. 1999] is similar to Nimble, but includes a more heterogeneous notion of component. Adaptors in their system are the default, rather than a special case: each component is accessed only through an adaptor. Adaptor logic is specified in a configuration file which can specify signature-level remappings. Unlike Nimble, procedural requests and responses are fully separated into effectively an asynchronous event-based model—the return path of a method call is modelled with a separate signature invoked by the callee, and may be separately adapted. Adaptors also control the implementation of event-based communication: they are responsible for constructing the system-level communication paths between each other at initialisation time. Auxiliary information in the configuration file can specify details of this (e.g. whether to use procedures, pipes or message queues).

LayOM [see Bosch 1999] is an implementation of a "layered object model" for C++, based on a concept of adaptation called "superimposition". Adaptation primitives are captured as operators known as *superimposing entities*, each defining a "layer" or black-box delta over the underlying object. Adaptations are implemented as deltas, and include member renaming, interface restriction, selective delegation (of field or method accesses), run-time configurable method dispatch, and interposition on field and method accesses. It improves on the expressivity of systems such as Nimble by separating out the *generic* nature of an adaptation from its *particular* uses in a given composition: existing operators can be specialised or combined by the user into new ones. New operators may also be defined from scratch, as a preprocessing stage for C++: the LayOM preprocessor can be extended with new syntax for the new superimposing entities, described by new lexing, parsing and code generation rules.

FLAME [see Eisenbach et al. 2007] is a tool for "flexible dynamic linking", targetting the Microsoft .NET Common Language Runtime. It performs a very restricted form of adaptation, affecting only system structure [see 4]. Bytecode for the CLR embeds not only the names of external classes, but also often em-

[4] Curiously, this paper also introduces a spurious distinction between "software adaptation" and "software adaption". The alleged distinction can be traced to a typographical error in their citation of [Bracciali et al. 2005]. The latter paper's correct title actually uses the typical word, "adaptation"; "adaption" is a little-used term which has, to this author's knowledge, never been explicitly differentiated from "adaptation" in any other work.

beds names of component implementations or "assemblies". The latter kind are not merely type names or package names (such as in Java), but are specific to individual implementations of those packages—for example, the name mscorlib is embedded by the Microsoft compiler into bytecode which consumes the CLR standard library. On other implementations of .NET, the library will have a different name, such as monolib [see 5]. Mismatches of these names prevent linking. FLAME modifies the compiler to embed metadata into the generated code, specifying which class or assembly names are to be treated as substitutable metavariables. Since the .NET runtime supports versioned libraries, FLAME's modified linker extends the existing version substitution support in the .NET configuration file format, so that it can also specify class- and assembly-name substitutions. The resulting adaptation capabilities are a proper subset of those of BCA.

Concept maps, a feature in the forthcoming revision of the C++ language, have been demonstrated [see Järvi et al. 2007] to resolve mismatch of data structures across the interfaces of *generic* libraries (i.e. libraries for template metaprogramming). Concepts provide a compile-time analogue of run-time subtype polymorphism: a concept is a constraint on a type variable, say T, as might be used in a template definition of the form template <typename T> .... A concept map is a declaration that a particular type satisfies a particular concept. If existing code for the type does not already satisfy the concept, a concept map may contain auxiliary code to specify *how* that concept may be satisfied using that type. To support run-time dispatching, for cases when the type used to instantiate the generic library interfaces is not known when compiling the client, concept maps may be used to adapt between early-bound nonmember functions and late-bound member functions. Since templates are elaborated statically, run-time performance is mostly identical to hand-crafted implementations. Local static variables can be used to implement protocol (stateful) adaptors within concept maps, using regular C++ code. In general, concept maps can be thought of as a syntactically convenient "adaptor pattern" analogue, applying to meta-level concepts rather than conventional interface types.

2.6 Linking and interconnection languages

Many researchers have proposed languages which give explicit control over the relationships between modules in a large system. These languages are usually called "module interconnection languages" or "linking languages". Often they are designed to be useful as high-level structural descriptions of a large system developed, frequently also embodying a novel development method or information-hiding technique [see Parnas 1972]. The first example of such a language was

[5] This is the name used in the Mono implementation; see http://www.mono-project.com/.

probably MIL 75 [see DeRemer and Kron 1975]. Clearly these descriptions are a convenient domain to adapt the *structure* of a system, as compared with manually renaming symbols within source code or binaries.

The Knit linking language [see Reid et al. 2000] confers full and explicit control over the linkage relation over a set of object files, by rewriting symbol names at link-time from a high-level description of object file instances and their intended linkage. It also supports hierarchical information hiding, and supports cyclical inter-module reference structures. While operating at the object code level, it optionally transforms the originating C source code for whole-program optimisation purposes. Jiazzi [see McDirmid et al. 2001] applies the same model and linking capabilities to the more complex case of Java linkage.

As already described, renaming is also the technique used by Flexible Dynamic Linking (see [Section 2.5]). DITools [see Serra et al. 2000] provides a dynamic analogue to Knit, supporting load- and run-time rebinding of *dynamically-linked* symbol references, as described in a configuration file. (Much as the adaptor pattern is a more flexible and more dynamic alternative to implementation inheritance, DITools offers such an alternative to such mechanisms as the `LD_PRELOAD` supported by Unix dynamic linkers.)

A surprisingly similar technique is "dependency injection" [see Fowler 2004]. This is implemented by certain "containers", such as Spring [see 6] or Castle [see 7], targetting enterprise application development for the Java or Microsoft .NET runtimes. Containers are run-time environments supporting an event-driven or "inversion of control" programming style. User code is purely reactive, and the proactive part of a program (e.g. an event loop) is provided by the container [see 8]. Dependency injection is an additional technique which eliminates the need for explicit references to implementation classes. Rather than explicitly instantiating classes in order to consume their services, the client programmer simply declares a field or constructor through which an object fulfilling the dependency can be passed at run-time. The fulfilment of these dependencies, including the choice of implementation classes, is described separately in a configuration file. This file is interpreted at load-time by the container. Like a linking language, the configuration language can be used statically to capture and alter the wiring between objects.

The first linking technology with extensive adaptation support was the combination of the OMOS linker and the Jigsaw language [see Bracha et al. 1993]. OMOS is a long-running linking service, while Jigsaw is a language of transformations over object files. The latter is founded on Seeley's earlier insight that module instances (such as instances of object files) and objects (as in object-

[6] http://www.springframework.org/
[7] http://www.castleproject.org/
[8] This separation between proactive and non-proactive is the same as that suggested by service-oriented computing (see [Section 2.11]).

oriented programming) may be unified [see Seeley 1990]. OMOS may be invoked not only on concrete modules, but also on "meta-object" descriptions, specified as transformations of existing concrete objects. Jigsaw defines an abstract data types for objects, including many operations used to transform and combine objects: symbol hiding, renaming, rebinding and copying, merging two modules, and others. These primitives are shown to support derived constructs such as functional interposition, although there is no support for modularising these derived constructs into more abstract adaptation functions (cf. LayOM), nor for finer-grained transformations (e.g. at argument level, cf. Nimble and the like).

2.7 Aspects, subjects and decentralised modularisation

Several technologies provide alternative ways to modularise large codebases, and in so doing, provide a domain which expresses some kinds of adaptation. Perhaps the most longstanding scheme used to modularise widely-scattered changes to codebases is the patch, as supported by Unix's patch command. Coccinelle [see Padioleau et al. 2008] generalises from this to provide a "semantic patch", with the aim of reliably and reusably capturing the semantic intent of a patch. Its target domain is the patchsets arising from evolution in large codebases, such as the Linux kernel, where development is continuous and decentralised. The system's semantic patch language SmPL uses pattern-matching to avoid manual specification of each source code location requiring modifications. Patches are merged with the main-line code ahead-of-time, as a manual process separate from compilation or loading.

Aspect-oriented programming is a relatively new and popular modularisation technique. Aspects in their most general form, as described by [Filman and Friedman 2000], are modules expressing modifications or additions to existing code in a way which supports *quantification* and *obliviousness*: points of application are identified by some logical expression *quantifying* over the existing codebase, and the existing code's developer may remain *oblivious* to these changes. Typically aspects are advocated as a convenient way of modularising cross-cutting concerns within a single project [see Kiczales et al. 1997], but the obliviousness property makes them useful for specifying adaptation.

AspectJ [see Kiczales et al. 2001] is the best-known implementation of aspects; it operates on source-level representations of Java programs. Points in a program's execution where control is transferred to aspect code are identified by quantifying expressions called *pointcuts*. This quantification is dynamic, in that it may be predicated on the program's execution context (e.g. on the class of the current method context's object). Pointcuts frequently embed particular class and method names, so are usually highly specific to a particular target codebase.

Conceptually similar to aspect-orientation is the idea of "subject-oriented programming" introduced by [Harrison and Ossher 1993]. Its goal is to enable the construction of new applications out of separate codebases, where this separation is motivated either by encapsulation concerns or by independent development. Codebases, or "subjects", share a common domain, but concern partially differing properties, operations and taxonomies (i.e. class hierarchy structures) of that domain's objects. The approach separates the *identity* of an object from the *multiple* collections of state and code which realise it concretely. Each subject may specify different instance variables and different method suites for a particular object, and arrange objects in a different class hierarchy. *Composition rules* are used to specify correspondences between the subjects' class hierarchies, instance variables, and dispatch strategies for method calls (since the latter may trigger execution of multiple subjects' code). [Ossher et al. 1995] proposes rules composed of essentially the same primitives as Jigsaw's [see Bracha et al. 1993], but allowing specification in the form of *general rules* (quantifying over all subjects, or classes, methods etc.) supplemented with *exceptions* for special cases.

2.8 Program transformation

Several projects have developed systems for transformation of Java programs. These are of similar spirit both to BCA (Binary Component Adaptation; see [Section 2.5]) and aspect-oriented Java (see [Section 2.7]), but support more powerful white-box adaptors.

JOIE [see Cohen et al. 1998] is a load-time transformation system. In it, "transformers" are Java-language components which inspect and modify existing class definitions using a reflection-like API. Transformers can be specified highly invasively, and are specified essentially as programs operating over bytecode (down to the level of instruction mnemonics). Unlike with AspectJ and other ahead-of-time techniques, there is no pattern language for identifying modification sites; regular Java-language iteration and if–else tests are required. There is a strong resemblance between JOIE transformers and COMPOST metaprograms (see [Section 3.2]). However, JOIE's level of abstraction is lower, being at the binary bytecode level rather than the source level.

Javassist [see Chiba 2000] similarly introduces an expanded reflection API to Java, designed as a general-purpose interface upon which to write tools such as BCA-like adaptation primitives, AspectJ-like pointcut or "hook"-based interposition rules, and load-time stub generation for remote method invocation systems. Unlike JOIE, its interface does not descend to the instruction level; rather, it provides only higher-level primitives, such as method wrapping and field access redirection.

JMangler [see Kniesel et al. 2001] is yet another project with similar goals. Its expressivity lies between those of JOIE and Javassist: unlike Javassist, it ex-

presses all transformations which preserve binary compatibility of Java bytecode, whereas unlike JOIE, it rules out transformations which break compatibility. It also addresses the unanticipated combination of multiple independently developed transformers, where a change specified by one transformer may trigger changes in others. This is done by introducing a distinction between *interface* and *code* transformations: the former are shown to yield an order-independent fixed point when applied iteratively, which can be done mechanically, whereas the latter's fixed points are order-dependent and therefore must be combined manually.

2.9 Software connectors

Architecture description languages (ADLs), much like linking languages, are designed to describe, explain and reason about large-scale structural properties of systems. They also promote re-use of high-level designs, and to enable checking and traceability between implementation and design [see Garlan and Shaw 1994]. As such, they not only offer the same structural view as that of linking languages, but have, in a few cases, introduced other features useful for adaptation—mostly concerning the notion of "connectors". These may be thought of as re-usable abstractions concerning *how* two modules might be connected—in other words, how they communicate. Traditional programming languages do not cleanly capture such abstractions, and implementation of communication mechanisms is typically spread across program modules and in tool-generated code [see Shaw 1994].

UniCon [see Shaw et al. 1995] was among the first architecture description languages to feature connectors [see 9]. UniCon resembles a module interconnection language, but introduces a distinction between components (which are C source files in the implementation presented) and connectors. Connectors are implementations of communication abstractions, such as local or remote procedure calls, pipes, real-time resource schedulers and shared variables. Components' binding points, or "players", are wired to the connectors' binding points, or "roles". The UniCon compiler uses built-in knowledge of each connector type to build the complete system, first generating intermediate artifacts (such as RPC stubs or makefiles) and later invoking a conventional build system. The system can be extended with new connectors through a slightly complex process which generates a new compiler [see Zelesnik 2000]. The ability to mix-and-match these communication implementations gives it a very limited additional domain of adaptation, above the structural domain offered by linking languages.

[9] The other, Wright [see Allen and Garlan 1997], was a complementary language developed to demonstrate a formal semantics for software connectors. Since it was of a less practical bias than UniCon, and had no special features directly supporting adaptation, I do not discuss it here.

COMPOST [see Assmann et al. 2000] explicitly combines adaptation techniques with connectors. Connectors in this system are defined by metaprograms which not only generate the necessary communicational code, but can also refactor or "rebind" the source code of existing components, written in Java or C++. First, existing code using method calls is rewritten into a generic "abstract" model of communication based on object exchange. Secondly, this abstract code is rewritten to use a new concrete communication mechanism, weaving in the necessary glue code. These rewriting procedures are expressed as metaprograms over Java or C++ abstract syntax—they are white-box [see 10] deltas over the input implementation.

The term "connectors" has been re-used by the coordination community. I will discuss this work in [Section 2.12], and contrast the two communities' interpretations of the idea in [Section 3.2].

2.10 Packaging

The term "packaging" was used by [Callahan 1993] to refer to the "details of how software configurations are 'packaged' into executables". These details include both programming-level details—data encodings, control flow patterns and APIs required by communication mechanisms—and lower-level details (such as `make` rules) for building binaries compatible with loading and linking mechanisms. Packaging systems may support whole-system description, like ADLs, or else only single-component packaging, but in either case clearly provide adaptation techniques.

Callahan's Polygen system [see Callahan and Purtilo 1991] accepts declarative descriptions of modules in several procedural languages, together with a composite system description, and uses a "rule base" to generate a makefile which can construct a complete system. The rule base contains knowledge about different procedural packaging styles—such as rules for invoking RPC stub generators or language-specific wrapper generators. Like the later UniCon, discussed in [Section 3.2], the system description can select different among different implementations of various communication abstractions, and new implementations can be defined (using rules written in Prolog). However, Polygen is tied to a procedural communication abstraction, and therefore can't adapt between different styles such as event-, stream- or dataflow-based communication.

Packaging has subsequently received attention from the software architecture community. As described in [Section 2.9], UniCon's notion of connectors, together with its ability to generate system implementations, give it essentially the same capabilities as Polygen. Later work considers the orthogonalisation of packaging from the implementation of functionality [see Shaw 1995]. DeLine's

[10] ... or grey-box, according to the authors' definitions.

system Flexible Packaging [see DeLine 2001], based on UniCon, provides support for composing more heterogeneous communication styles at a packaged component's interface, at the cost of requiring that a raw component (or "ware") is programmed to a channel abstraction. UniCon is used to describe packaging requirements, with extensions to describe details such as syntax of pipeline data.

2.11 Orchestration and service-oriented architectures

Several technologies with relevance to adaptation have emerged from the World-Wide Web. I will discuss three: orchestration, XML-based data transformation languages, and service-oriented architectures.

The word "orchestration" often refers to scripting of network-enabled services, most typically Web Services. Owing to the widely distributed nature of such services, orchestration languages such as BPEL [see Peltz 2003] and Orc [see Misra and Cook 2006] provide special features relating to concurrency, latency and failure. For example, Orc provides a parallel composition operator, support for pipeline-like streamed continuous communication, and both pruning (as a feature) and timeout (as an idiomatic derived form) for late-responding services. Orchestration may therefore be seen as scripting tailored to wide-area distributed execution [see 11].

As with scripting, orchestration's main contribution to adaptation is its provision of a domain, separate from component code, which centralises integration details and is convenient for adaptation by source-level edits. Orchestration languages might also usefully integrate adaptation primitives, such as analogues of the regex-based rewriting seen in scripting languages. To this author's knowledge, no current orchestration languages have these features. However, exactly this kind of rewriting function is provided by technologies such as XSLT [see 12] and XQuery [see 13]. Both were developed to operate on XML documents: the former as a customisable specification language for XML-to-HTML prettyprinters, and latter as a query language. Both have grown into Turing-complete languages [see Kepser 2004] with useful adaptation facilities such as projection, renaming and (in XSLT's case) pattern-based rewriting of tree structures.

Service-oriented architectures are software architectures which decompose a system into passive "services" (such as web services) and proactive components (as might be written in an orchestration language). It is generally agreed that service-oriented architectures aim to *integrate* separately developed applications, without rewriting them from scratch [see Channabasavaiah et al. 2003; Pisello

[11] Indeed, there are many similarities between orchestration and recognised scripting languages. For example, the Unix shell has special features for parallel and redundant execution in its `&` and `||` operators.
[12] http://www.w3.org/TR/xslt
[13] http://www.w3.org/TR/xquery/

2006]. Two other properties of service-oriented architectures are commonly cited: service interfaces are constrained to be "uniform" in some sense, and *requests* to a service (e.g. procedure calls) are encoded in a self-describing extensible form [see He 2003]. "Loose coupling" is a commonly claimed consequence of this design [see Papazoglou 2003; He 2003]. Despite these goals and claims, no specific adaptation techniques have emerged from service-oriented technology. Rather, the migration process—of remodularising existing systems into reactive re-usable *services* and proactive, process-specific *orchestrations*—is implicitly to be done manually and conventionally. I will return to the idea of "loose coupling", and the uniform interfaces technique, in [Section 3.3].

Orchestration is often seen as a special case of *coordination*. The next section will discuss coordination more generally.

2.12 Coordination

Coordination is notoriously difficult to define, but might be described as the composition of components under a special awareness of parallel execution, synchronisation and scheduling constraints. It has been described as "managing dependencies between activities" (cited by [Papadopoulos and Arbab 1998]), by [Gelernter and Carriero 1992] as "gluing together of active pieces", and by [Wegner 1996] as "constrained interaction" . The word "interaction" is often used to describe the domain of coordination, and I consider this word technically synonymous with "communication".

Many coordination languages, such as Linda [see Carriero and Gelernter 1989] and its many variants [see Papadopoulos and Arbab 1998], are simply special communication interfaces embedded into conventional programming languages. Linda provides a simple set of input and output primitives which manipulate tuples in a shared memory abstraction called a *tuple space*. Ongoing computations also appear in this space—this gives orthogonal treatment of already-computed and to-be-computed tuples, and hides synchronisation from the programmer. However, Linda is not convenient as an adaptation domain. In particular, its nondeterministic input semantics make precise interposition impractical.

Exogenous coordination [see Arbab 1998] techniques, however, *are* useful for adaptation. Exogenously coordinated components never interact directly, but instead exchange opaque messages with a coordination engine, which is responsible for routing, synchronisation and scheduling concerns. In Reo [see Arbab and Mavaddat 2002], the behaviour of this engine is specified as a network of channel primitives with specified synchronisation and data-flow behaviours. Not only do the connectors express the linkage relation between components, but they enable changes to the concurrent execution, synchronisation and scheduling behaviour of the system. For example, the network of connectors may be

modified, independently of components, to prevent deadlock or improve parallelism. This permits component aggregation, interposition and also protocol adaptations. However, apart from filtering, Reo's coordinators do not inspect or modify the messages themselves, so any adaptation of messages must be done in a manner opaque to Reo, by adding or changing components. (Combining Reo with aspect-oriented techniques has been proposed by [Eterovic et al. 2004], using pointcuts to exogenously instrument components with channel endpoint logic; one could also add message adaptation logic in this way.)

2.13 Systems performing specialised adaptation

The practice of software adaptation far predates research into general or principled approaches. Many specialised adaptation techniques have been developed to answer particular application needs. Each of the following examples is usually considered in isolation as its own technique, but I show that each is simply a particular kind of adaptation or adaptor.

Virtual machine monitors (VMMs) are one kind of special-purpose adaptor [see 14]. Operating system kernel binaries are written to execute in the privileged (kernel) mode of the underlying instruction set architecture (ISA). Interposing a virtual machine monitor requires the kernel to run in the unprivileged mode (user mode), which provides a somewhat different interface, so the VMM must perform adapt between these. Some ISAs permit black-box "unmodified" virtualisation, since they generate traps when privileged instructions are executed by the virtual machine. Other ISAs, including Intel's x86, preclude this, since certain instructions with differing user-mode semantics *do not* generate traps [see 15]. Two techniques addressing this are binary rewriting [see Devine et al. 2002], as adopted by VMware, and paravirtualisation as adopted by Xen [see Barham et al. 2003]. The former rewrites binaries at run-time to replace the problematic instruction sequences with calls into the VMM; the latter uses ahead-of-time manual source-code edits so that the resulting binary contains no such sequences. While being merely manually-coded adaptors, rather than *techniques* for adaptation, VMMs have been used as an interposition mechanism to enable several related adaptations, e.g. for device driver re-use [LeVasseur et al. 2004] or software device implementations [Whitaker et al. 2004].

---

[14] One might also consider "virtual machine" language runtime implementations, such as those of Java or Microsoft's CLR [Meijer 2002], as adaptors. I do not, for the following reason. There is usually a considerable abstraction gap between the virtual machine architecture these systems provide, and the real machine architecture they are built upon. This contrasts with virtual machine monitors, which provide an interface little more abstract than that of the underlying machine. This "semantic gap" test is a plausible one for determining whether a piece of implementation is more like an adaptor, or more like a regular component. Of course, there is neither a rigid distinction, nor a need for one.

[15] Intel has since extended x86 with support for unmodified (hardware) virtualisation, although only after the popularisation of the two techniques described.

notations (see [Fig. 1])	$N = \{I, C, S, M, \dots\}^{1,2}$
edit to a notation n	$Edit(n)$
adaptor selection from a set s	$ASel(s)$
adaptor definition in notation n	$ADef(n)$
adaptor synthesis from notation n	$ASyn(n)$
yielding an adaptor of kind...	$\rightarrow$
container applying to notations $s \subseteq N$	$Cont(s, b \subseteq \{\blacksquare, \square\})^3$
delta applying to notations $s \subseteq N$	$\Delta(s, b \subseteq \{\blacksquare, \square\})^3$

[1] named languages may also be included

[2] for notation n, $n^{+\{e\}}$ means n extended with the set of features e; n^{-} means a constrained form of n

[3] $Cont_n$ and Δ_n refer to containers or deltas applying to exactly n components

Table 1: Concise language for modes of adaptation

Wrapper generation for programming language interoperability is another example of specialised adaptation. Tools such as Swig [see Beazley 1996] adapt a module written in one language such that it can be consumed by another. The process is parameterised by rules controlling how features of one language should be mapped to those of the other. This permits some flexibility in how each module concretely captures the interface of the other, and can therefore be used for adaptation, although it is unlikely to be sufficient to avoid further manual glue coding. Many languages' foreign function interfaces, such as Java's JNI [see Liang 1999] and Haskell's FFI [see Chakravarty et al. 2002], provide similar wrapping capabilities, but offer still less flexibility.

Network proxies or middleboxes have long been used to alter or extend the features of a network. Recent work has applied this idea in order to adapt applications for mobile use under intermittent connectivity [see Kouici et al. 2005]. The various successes and limitations of such systems emphasise the influence of system infrastructure design—particularly the available points of interposition—on the space of feasible application-level adaptations.

Stub generation in RPC systems [see Birrell and Nelson 1984] or, more recently, in Web Services and Remote Method Invocation [see Waldo and Clemsford 1998], is another domain-specific form of adaptation. The automatically generated stubs (for the client-side invocation) and skeletons (for the server-side dispatch) are adaptors from local procedural communication, passing messages on the stack, into a distributed version of the same, passing messages over some network socket. They are mostly black-box, although some implementations may force client code to add extra error handling, for errors associated

name of system or author	**target representations adapted** (implementation, unless stated)	**domains** (general)	**representation of adaptations** (specific)	**modes of adaptation** (general)	**primitive adaptations** (specific)
adapter pattern	object-oriented code	message sequencing type metadata implementation	object or class definitions in the implementation language	$ADef(I) \rightarrow Cont(I, \blacksquare)$	any expressible within implementation language
Nimble	procedural code (multiple languages)	message	Ada-like pattern-mapping language	$ADef(D) \rightarrow Cont(I, \blacksquare)$	argument string rewrite
Yellin & Strom	procedural or OO code with finite-state behavioural specifications	message sequencing	(FSM, synthesised from partial specification consisting of message mapping rules)	$ASyn(S) \rightarrow \Delta_2(I, \blacksquare)$	complete synthesis of adapter, between pair of interfaces
Bracciali	procedural or OO code with pi-calculus behavioural annotations	message sequencing structure	(pi calculus process, synthesised from partial specification consisting of event mapping rules)	$ASyn(S) \rightarrow \Delta_2(I, \blacksquare)$	complete synthesis of adapter, between pair of interfaces
AxML	purely functional ML code, with semantic specifications	message	(ML expression synthesised from specification axioms)	$ASyn(S) \rightarrow \Delta(I, \blacksquare)$	complete synthesis of structure/function definition
Concept maps	C++ generic libraries	message sequencing metadata	C++ concept map definitions	$ADef(I) \rightarrow \Delta(I, \blacksquare)$	addition of C++ code, to satisfy some individual requirement of a concept
Binary Component Adaptation	Java bytecode	structure typing metadata	simple "delta language", Java-like	$ADef(I^-) \rightarrow \Delta(\{I, M\}, \blacksquare)$	renaming classes, methods, fields rewriting references to classes, methods, fields method addition interface clause addition
LayOM	C++ code, with or without existing use of LayOM extensions	message structure + others possible by extension	superimposition expressions in LayOM C++ extensions + user-defined C++ preprocessors	$ADef(I^{+\{ext\}}) \rightarrow \Delta(I, \{\blacksquare, \square\})$ $ADef(I^-) \rightarrow \{ext\}$	interface member renaming, restriction (client- or state-dependent), message delegation, component aggregation, functional interposition + others by extension
FLAME	implementation: CLR bytecode (annotated) configuration files	structure	rebinding clauses within application configuration files, XML syntax	$ADef(C) \rightarrow \Delta(I, \blacksquare)$	rewriting references to classes, assemblies
Knit	implementation: object files configuration: Knit linkage description	structure	changes within Knit language, hierarchical module descriptions	$ADef(C) \rightarrow Cont(I, \blacksquare)$ $Edit(C)$	rewiring (symbol renaming) hierarchical combination of modules interface restriction (hiding)
Jigsaw	object files	structure	Jigsaw meta-object expressions, s-expression syntax	$ADef(S) \rightarrow \Delta(I, \blacksquare)$	module merge symbol overriding interface restriction symbol renaming
AspectJ	Java source code aspect definitions	message structure implementation	extensions of Java language: pointcut pattern-language and near-arbitrary Java code for inserted "advice"	$ADef(I^+) \rightarrow \Delta(I, \square)$	context-dependent code interposition and replacement, on method calling/called, field get/set, initialisation, exception handler entry
JOIE	Java bytecode	message structure implementation typing metadata	Java classes implementing ClassTransformer interface, consuming defined library interface for bytecode metaprogramming	$ADef(I^-) \rightarrow \Delta(\{I, M\}, \{\blacksquare, \square\})$	renaming, addition, removal of classes, methods, fields altering access modifiers, declared thrown exceptions instruction splicing rewriting references to classes, methods, fields modifying subtyping relation

Javassist	Java bytecode	message structure implementation typing metadata	Java classes consuming defined library interface	$ADef(I^{-}) \rightarrow \Delta(\{I,M\},\{\blacksquare,\square\})$	renaming classes, methods, fields altering access modifiers, declared thrown exceptions rewriting references to classes, methods, fields field/method addition, removal modifying subtyping relation
JMangler	Java bytecode	message structure implementation typing metadata	Java classes consuming defined library interface	$ADef(I^{-}) \rightarrow \Delta(\{I,M\},\{\blacksquare,\square\})$	method and field addition, renaming changes to subtyping relation preserving binary compat. changes to method code composition of some interface transformations altering "throws" clause
UniCon	implementation: procedural code (multiple languages) configuration: UniCon system description	structure implementation (chosen among pre-sets)	structural changes and connector substitutions within UniCon configuration language (has both textual and graphical syntaxes)	$ADef(Perl^{-}) \rightarrow Cont(I,\blacksquare)$ $Edit(C)$	connector rebinding (rewiring) substitution of compatible connector implementations (e.g. RPC for local procedure)
COMPOST	implementation: Java source code or C++ source code configuration: as implementation, but using special configuration API	implementation structure + others possible by extension	code generators consuming metaprogramming utility library and external tools (e.g. IDL compiler for "Corbify" example)	$ASel(\{metaprog\}) \rightarrow \Delta(I,\square)$ $ADef(I^{-}) \rightarrow \Delta(I,\square)$	"portify" (transform code from method-call form into object-exchange form) library of standard metaprograms (Corbify, Observify) + user-supplied metaprograms
Polygen	implementation: procedural or functional code config.: declarative system descr. spec.: declarative module descriptions	structure implementation (connectors only)	changes within declarative system description addition of packaging rules for new implementation of procedure call	$ADef(Prolog^{-}) \rightarrow Cont(I,\blacksquare)$ $Edit(C)$	mix-and-match of different packaging types (each being an implementation of the procedure call abstraction)
Flexible Packaging	implementation: Ciao (C code using channel I/O primitives), specification: UniCon packagers (wrappers)	message sequencing implementation (of packager, constrained)	changes within UniCon packaging descriptions invoking build tool using changed selection of packaging definition	$ASel(\{packager\}) \rightarrow Cont(I,\blacksquare)$ $Edit(C)$ $ADef(C) \rightarrow Cont(I,\blacksquare)$	mix-and-match of pre-fabricated packagings any legal edits to UniCon packaging description
subject-oriented composition	implementation: compiled object-oriented code, with label annotations configuration: composition rules	message structure	rules specifying class, method and instance variable matchings, and method dispatch strategies exceptions to existing composition rules	$Edit(C)$ $ADef(C) \rightarrow Cont(I,\blacksquare)$ $ADef(C) \rightarrow \Delta(I,\blacksquare)$ $ADef(C) \rightarrow \Delta(C,\square)$	selective merging and overriding of separate subjects' class, variable and method definitions
Coccinelle	impl.: C source code spec.: semantic patches (SmPL)	implementation	abstract syntax-level rewrite rules	$ADef(S) \rightarrow \Delta(I,\square)$	arbitrary source code changes
scripting languages (using Piccola example)	implementation: components in style-dependent packaging (e.g. Java AWT components) configuration: scripts	message sequencing structure (metadata in some cases)	changes within scripts	$Edit(C)$	any permitted by scripting language
Reo	implementation: send-receive components implementing configuration: Reo circuits	sequencing structure	changes within Reo circuits	$Edit(C)$	channel addition, removal, type modification node addition, removal

Table 2: Summary and comparison of surveyed practical adaptation systems

with distributed execution.

2.14 Summary

To concisely describe the *modes of adaptation* of each technique, a concise language is given in [Tab. 1]. This is intended as a concise and familiar notation, interchangeable with the terms shown in [Fig. 1]. It does *not* imply any underlying mathematical formalism.

Using this language, [Tab. 2] summarises the bulk of the adaptation systems surveyed, with reference to the criteria outlined at the start of the survey.

3 Review of design principles

It is difficult to measure the value or effectiveness of an adaptation technique. We have seen a wide variety of techniques, each occupying a particular point in the design space. In this chapter I critically examine some cross-cuts of the material surveyed, attempting to refine out various different implied notions of what constitutes a *good* adaptation technique.

3.1 Component models

Perhaps the most-used term when considering software re-use is "component", which originates with [McIlroy 1969]. Each adaptation technique has its own constraints on *what* representations of software it can adapt: these constraints constitute its *component model*. All systems performing composition of software, whether or not they provide special adaptation techniques, have some kind of *component model*. Sometimes, as with component middlewares such as Enterprise JavaBeans, COM+ or CORBA, the model is explicitly termed as such, and standardised in a document. At others, such as with most conventional programming languages and operating systems, it is defined using different words, such as "module system" or "linkage model".

Most authors' implicit perception of a good component model is one which, when combined with the adaptation techniques applicable to it, minimises the cost of developing a complete system out of independently evolving and/or independently developed components. Therefore, the goodness of a component model is dependent on the goodness of adaptation techniques which are applicable to it, and vice-versa.

So far I have written as if any notion of component, however defined and whatever its properties, constitutes a component model. However, many authors reserve the term "component model" for a subset of these notions, i.e. those meeting some minimum criteria. In this section we analyse these criteria as a source of wisdom about what makes a *good* model, with reference to some of the adaptation techniques surveyed in the previous section.

3.1.1 Component middleware

Industrial adoption of the term "component" followed the popularisation of object-oriented programming techniques in the 1980s and 1990s. [Szyperski 1998] defines a component as "a unit of composition with contractually specified interfaces and explicit context dependencies only… [it] can be deployed independently and is subject to composition by third parties". Meanwhile, another definition is proposed by [Councill and Heineman 2001]: "a software component is a software element that confirms to a component model and can be independently deployed and composed without modification according to a composition standard" [see 16].

There are several consequent good properties of such models. Explicit "*requires*" interfaces, symmetric with explicit "*provides*" interfaces, are a form of information hiding in the sense of [Parnas 1972]. They ease integration into a new environment, by helping third parties to identify a mutually satisfying selection of components and any necessary adaptation logic.

Heineman and Councill's definition includes two requirements not stated by Szyperski. One is some kind of homogeneity: something is a component only if it conforms to a well-defined *standardised* component model, where standardisation is implicitly a tool for cutting down on the extent of necessary adaptation. It does so by reducing the potential for diversity among candidate components, at the cost of narrowing the field of such candidates. (This makes it a *coupling mitigation* technique—see [Section 3.4.3].) This definition echoes the design choices behind systems targetting relatively more homogeneous component implementations, such as C++ generic libraries (in the case of concept maps [see Järvi et al. 2007]) or Java classes (in the case of BCA [see Keller and Holzle 1998], JOIE [see Cohen et al. 1998] and similar systems).

The second additional requirement is that component composition treats each component as a *black-box*, i.e. that components are composed "without modification". This echoes the benefits claimed by *non-invasive* adaptation systems, exogenous coordination and the like, which frequently observe that non-invasive adaptation is cheaper than invasive adaptation [see Bosch 1999; Arbab 1998; Purtilo and Atlee 1991; Keller and Holzle 1998].

Are components static or dynamic? The two definitions above agree that "components" is to mean "component implementations": the units of implementation and deployment, firmly static entities. Meanwhile, *component instances* are the corresponding dynamic artifacts. This distinction would appear to be one of terminology: preferring the term to mean "component instances" (roughly equivalent to "objects") can also be justified, and consistency is more important than the choice itself.

[16] "Composition standard" and "component model" are defined, albeit slightly loosely, elsewhere in the same chapter.

However, since we are considering adaptation, and have established that any component model is implicitly partnered with a set of applicable adaptation techniques, a closely related distinction *does* become important. Some adaptation techniques operate on dynamic objects (e.g. the adaptor pattern, aspect-oriented programming), and the meaning of the adaptation performed can depend on dynamically evaluated properties of these objects. Meanwhile, others operate only on static artifacts (e.g. Knit, FLAME, and most other binary-level techniques). Among the latter, there is an additional related distinction: whether adaptation is applied on a per-instance or per-implementation basis. For example, Knit and Jigsaw allows multiple instances of each module, and each can be composed differently, even though composition is performed statically. Meanwhile, systems such as FLAME and the various patch-based tools do not offer this discrimination: they clearly affect all instances at once [see 17]. In systems that provide it, this discrimination is usually cited as a benefit; without it, adapting different instances differently would most likely require manually duplicating the component implementation, and incur associated overheads.

In summary, we have observed a preference for the ability to distinguish between multiple *instances* of a component, whether statically or dynamically. We will revisit the "static versus dynamic" issue in the next subsection.

3.1.2 Software architecture

Typically components are defined in the software architecture literature as things which "roughly correspond to compilation units of conventional programming languages" [see Shaw 1994]. Expanding on this "architectural" view of components (and connectors), [Allen and Garlan 1997] give us insight into whether components should be considered as static or dynamic artifacts.

They describe a toy system *Capitalize*, which transforms an arbitrary alphabetic character stream by outputting it in alternately lower- and upper-case characters. The system is a network containing four computational components and four pipes. From the diagram, we see that at the architectural level each pipe is distinct, despite being implemented by the same code.

They do not describe the precise differences between the "implementation" and "architectural" notations, but there are four: logical containment (shown by the larger Capitalize box enclosing the other features); abstraction by elimination of uninteresting modules (shown by the absence of a *config* component); first-class dynamic objects (shown by the presence of four distinct pipes, instead of connections to the I/O library); and finer-grain interfaces (shown by the distinction between the input and output ends of the pipes). [Fig. 2] shows the

[17] Note that in the dynamic case, this is not an issue: systems such as aspect-weavers, which can identify target component instances using dynamic properties, invariably include object identity as one such property.

effect of progressively adding these notational features.

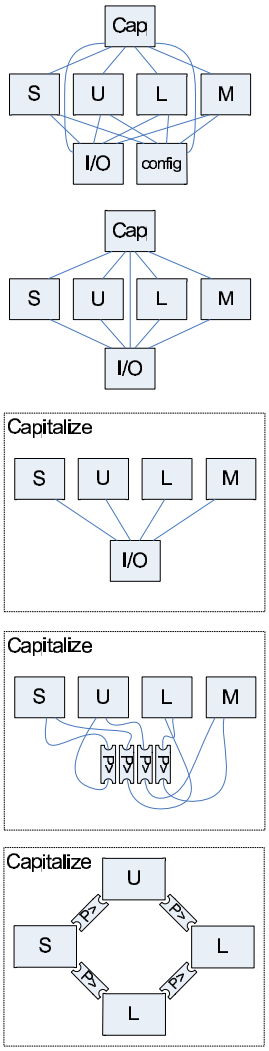

Figure 2: From implementation to architecture.

Clearly, dynamic objects are considered architecturally significant, whereas the static views of the system (first three pictures) hide this significant information. This is in agreement with the designs of Knit and Jigsaw, since it implies that an architectural notion of components (and connectors [see 18]) should ac-

[18] Although the pipes in the example are termed "connectors" rather than "compo-

cept that component *instances* are significant, rather than the identity of their implementations—the *meaning* of a component derives from its participation in a running system.

In summary, we have observed that the significance of a component instance need not be shared by other instances sharing the same implementation. Consequently, although we may perform adaptation ahead of time, this should not affect our ability to differentiate individual component instances when doing so. Additionally, since, in general, component instances may be created dynamically, ideally any adaptation logic's ability to distinguish instances should extend to dynamically created ones. Although this ability is found in systems such as AspectJ, where a pointcut may be predicated on dynamic properties of the current object, most systems we have seen do not provide it. However, since many adaptation tasks will not require this dynamic ability, and since it most likely incurs run-time overhead, its absence can be justified.

3.1.3 Programming language research

In programming language research, subtle details distinguish one module system from another—for example, the mathematical properties of their type systems. These distinctions have been used to draw "component" versus "not component" distinctions between modules. [Owens 2007] defines a component as "a standalone entity (with respect to other components) that can be shipped to another developer who can use it, consistent with its interface, without knowledge of the encapsulated details of its implementation" and further states that "a component can be implemented, compiled, and deployed independently of other components that might link with it". The requirement for independent compilation introduces a distinction between *components*, which support *external linkage*, and less well-encapsulated modules which do not qualify as components, since they support only *internal linkage*.

In other words, this definition argues that a *good* component model will be one in which components are required to support external linkage. This resonates with certain adaptation techniques surveyed earlier, including FLAME [see Eisenbach et al. 2007] and Jiazzi [see McDirmid et al. 2001], whose adaptations are designed to directly circumvent the problems caused by existing development platforms (respectively Microsoft's CLR and Java) embedding names from neighbouring components into generated code. A good component model would not suffer any need of such adaptations.

The above definition seems to return us to a static notion of components, in which components are units of implementation, compilation and deployment.

nents", this is immaterial: consider multiple instances of a hash table or some other non-shared data structure, each of which might also have its own architectural significance.

However, clearly *linking* with a component—the activity of concern—entails either *instantiating* it anew within the system image being constructed (as when linking with a library), or gaining a reference to an existing instance outside that image (as when connecting to a network service). Making a *static* requirement that a component be independently compiled, linked and deployed, is simply a conservative means of ensuring that these properties hold *for all* instances, including those created dynamically. Therefore, we should not consider this view to oppose earlier arguments emphasising the significance of instances over implementations.

3.1.4 Communication primitives

We have so far observed several connections between adaptation techniques and particular definitions of the *class* of component models. We would also like to examine *specific* component models, since any useful adaptation system ought to be applicable to at least some of these. [Tab. 3] lists some computational abstractions (or component models) and their communication primitives [see 19].

Considering real component models highlights two *practical* issues of component model design, both concerning implementability and checkability.

Homogeneity Different adaptation techniques assume different degrees of homogeneity among their components. This trades off two benefits. On the one hand, less homogeneity means potentially more re-use, since there are more candidate components. On the other hand, the technique can make fewer assumptions about the *behaviour* and *implementation* of components. The latter rules out techniques such as metaprogramming, while the former brings the challenge of a sufficiently expressive black-box model of software. Such a model would ideally capture the entire range of primitives found in the table. In general, less homogeneity makes it harder to implement adaptors and to check compositions.

Elegance versus precision Some abstractions, such as the Unix process, offer a wide variety of non-orthogonal communication primitives. Others, such as the pi-calculus, offer only a small and elegant set. Which are better for writing easily adaptable components? Elegance correlates with orthogonality and minimality, and hence with the absence of arbitrary distinctions which might prevent two compatible components from being combined. (Contrast this with Unix, where a somewhat arbitrary choice of IPC mechanism can

[19] Here I am implying that a computational abstraction is a component model if and only if it can express not only Turing machines but also "interaction machines" in the sense of [Wegner 1997]. In other words, it includes a notion of the "environment" or "outside", with which a component can communicate.

Computational abstraction	Communication primitives
generic instruction set architecture (ISA)	register file access main memory access shared I/O device register access program counter increment branch instruction indirect branch instruction interrupt
ISA-level virtual machine (e.g. Xen domain)	register file shared storage local memory access shared memory access program counter increment branch instruction indirect branch instruction hypercall event or upcall from hypervisor software interrupt from user-space upcall to user-space
C language	global variable access heap access arbitrary memory access statement sequencing function call indirect function call longjmp
Pascal language	global variable access heap access arbitrary memory access statement sequencing function/procedure call indirect function/procedure call
Unix process	virtual processor (sequencing, jumps, branches, registers etc, as in host ISA) virtual memory access (local) virtual memory access (inter-process) virtual memory access (trap to kernel) filesystem access sockets access other system calls signal handling process replication (fork) process replacement (exec)
Java language	static field access instance field access (through heap) static method call instance (virtual) method call exception handling
Haskell language	call-by-name evaluation
pi calculus	synchronous rendezvous (reduction step)
Reo component	read take write
C++ template metaprogram	class template instantiation class template specialisation function template instantiation function template specialisation
COM+ component	method invocation context inspection Win32 system calls

Table 3: Communication primitives of selected component models

hinder composition with components which chose alternatives.) Static checkability is also a benefit, for revealing compositions which are not correct. The two are somewhat at odds, since simpler models tend to offer less semantic discrimination between different kinds of communication, making it harder to distinguish correct from incorrect compositions [see 20]. I claim that we

[20] This argument is essentially the same as for pluggable type systems [see Bracha

would like to find a point along this trade-off which is optimal with respect to *coupling*. We return to this idea in [Section 3.3].

3.1.5 Summary

By examining definitions of component models, we have seen a broad agreement that adapting *instances* is preferable to adapting entire shared *implementations*; that orthogonalising the static or dynamic nature of components is desirable; and that adapting components without manual invasive modification is preferable. We have observed trade-offs in implementability and checkability, concerning the homogeneity of components and elegance (or orthogonality) of their communication primitives.

3.2 First-class connectors

If we understand the usefulness of adaptation, we understand the need to capture details about *how* components fit together. All the surveyed work has some particular mechanism for, or description of, this fitting-together. A similar concept, referred to either as "explicit connection" or the similar "first-class connectors", is popularly advocated in the literature of both software architecture [see Shaw 1994; Allen and Garlan 1997; Mehta et al. 2000] and coordination [see Arbab and Rutten 2003; Barbosa and Barbosa 2004]. Here I will review the origins, meaning and interpretation of the concept, and describe how it relates to adaptation. Later I will attempt to refine out precise statements of its intent and benefits, from what has become, in parts, a somewhat muddled literature.

Before proceeding, I remind the reader that the "confusion" described in this section does not imply that the concept of connectors in any one particular tool, paper or strand of work is at all confused. Rather, I am assessing the *different interpretations* of the term, across multiple (yet mutually referencing) research communities and technologies. These differences have made the research discourse less transparent than it might otherwise be.

3.2.1 The concept of connectors

The term "connectors" arose from the software architecture community. [Shaw 1994] provided the first published conceptualisation. Her paper observes that conventional programming languages are poor at expressing inter-module relationships and at localising the details of inter-module communication. These observations are sound and uncontroversial, today more than ever. The paper goes on to argue for the "first class" consideration of connectors as a logical

2004].

peer of components. It provides no precise definition of connectors, but plenty of examples and intuitions.

Later work by [Allen and Garlan 1997] and [Mehta et al. 2000] makes further contributions to the conceptualisation of connectors. However, the meaning of "connectors" is still far from uniformly understood, and the term is sometimes used in confusing ways. For example, in the coordination community, connectors are sometimes described as a particular kind of coordination primitive [see Arbab and Mavaddat 2002; Barbosa and Barbosa 2004], whereas according to Mehta's taxonomy, coordination is only of several possible roles of a connector.

A related problem is that the term has spawned an unhelpful dogma. Some work such as Darwin [see Magee et al. 1995], which doesn't make explicit distinction between components and connectors, is often criticised for (in the words of [Mehta et al. 2000]) "obscuring the distinct nature" of connectors. This is despite the fact that Allen and Garlan acknowledge in their formalisation [see Allen and Garlan 1997] that the distinction is one of practical convenience rather than logical essence (§11.3, p.241).

In the following subsections I discover a more precise characterisation of connectors, based on *mechanism* and *agreement*, and relate the concept to both coupling and adaptation.

3.2.2 Mechanism versus agreement

The examples of connectors listed in Shaw's original paper, and in Mehta's taxonomy, seem intuitively to be many different kinds of thing. They may or may not have independent identity at run-time. They may be implemented in user code, or within the programming language or operating system. They may, it is claimed, simply be abstract conventions, or may be complex pieces of implementation. They may connect statically known sets of components, or may form their associations dynamically. If connectors really are a coherent class of entity, it should be possible to characterise them succinctly.

One unquestionable property of a connector, or specifically of a *connector instance*, is that it provides a *communication mechanism* to one or more components within the system which contains it. A clearer phrase retaining the sense of "connector" might be "communication abstraction". Communication abstractions are like computational abstractions in many ways—they have abstract interfaces, and may have many distinct implementations. However, communication abstractions must join multiple parts of a system—perhaps multiple objects, multiple processes or multiple machines. *Agreements* on communication conventions are therefore necessary in order for these disparate parts to communicate successfully [see 21]. However, clearly these agreements by themselves are not communication mechanisms.

[21] The word "successfully" here differentiates the exchange of *meaning* from an ex-

This immediately eliminates some examples of connectors given by Mehta and Shaw. Shaw mentions shared data encodings, such as Rich Text Format, and other conventions; while being noteworthy, are not themselves connectors. Similarly, some examples in Mehta's taxonomy (e.g. X400, SQL) are agreements rather than connectors. Importantly, the noteworthiness of agreements is as the source of *coupling*, which we will discuss in [Section 3.3].

3.2.3 Connector types, instances and state

Another reason for the apparent diversity of connectors concerns the distinctions between connector types, connector instances, and their state in a running system. Shaw observes that connectors "manifest themselves as table entries, instructions to a linker, dynamic data structures, system calls, initialisation parameters, servers…". Most of these refer to the *state* of a connector instance rather than to its *type*, *identity* or *implementation*. Meanwhile, Mehta's taxonomy contains elements such as *inter-thread*, *inter-process*, *FCFS* and *LRU*—these are clearly abstract properties of communication mechanisms, rather than mechanisms in themselves. I consider these properties as part of the abstract *type* of connectors, i.e. some partial specification of a communication mechanism, which may include both its behavioural properties and its programming interface. This is completely analogous with the type of a component or module.

3.2.4 Illusory distinctions

Shaw mentions some apparent distinctions between components and connectors which, on further examination, reveal themselves to be illusory. One is the suggestion that components have *interfaces* while connectors have only *protocols*. In fact, they both have interfaces, and in either case those interfaces may incorporate protocol constraints. For example, interfaces to data structures may contain protocol constraints (e.g. that a stack may not be popped more times than it is pushed). This idea is now familiar from work on protocol adaptation [see Yellin and Strom 1997; Reussner 2003; Passerone et al. 2002].

An unarguable similar statement would be that while components and connectors may both have interfaces *and* protocols, it should be possible to modularise definitions of communication abstractions such that knowledge of *protocol state* is kept out of clients as far as possible. For example, an application which communicates using TCP sockets is not responsible for maintaining TCP protocol state, and its awareness of that state is confined to the simple abstraction

change simply of *information*, as distinguished by Weaver's statement [see Shannon and Weaver 1949] that "two messages, one of which is heavily loaded with meaning and the other of which is pure nonsense, can be exactly equivalent … as regards information". Effectively, agreements are agreed *coding rules* which give meaning to transmitted information.

offered by the sockets interface—which exposes states such as opened, listening and closed sockets, but does not expose the entire TCP state space.

Similarly, although Shaw observes that connectors appear not to have *typed* interfaces (considering the untyped byte-stream abstraction offered by Unix pipes), in fact it is simply that they may only be captured by parameterised *polymorphic* types. Although particularly common for communication abstractions, this is also observed in many conventional components, such as generic data structures.

3.2.5 Adaptors and connectors

Adaptors clearly enable communication between components. Therefore, they may be seen as some kind of connector. How can we distinguish them from other connectors? Since information flow is directional, agreements associated with connectors typically come in complementary or "handed" pairs: caller–callee, publisher–subscriber, producer–consumer, and so on. This pattern clearly extends down to the level of data structures: a caller who provides an argument tuple expects a callee who can accept that same tuple structure. Adaptors are necessary when these complementary relationships are broken: when different tuple structures must be consumed than those that are produced, or where the roles understood by the various participants in the communication do not match up. Adaptors may even be considered a *most general* form of connectors, in that a non-adapting connector is performing a trivial no-op or null adaptation.

3.2.6 The connector–component continuum

Some pieces of software clearly perform no communication, and therefore cannot be connectors. For example, consider a completely stateless library, such as a mathematical library, or a storage component with only one client component. However, in any case involving shared state, it becomes impossible to *intrinsically* distinguish computational from communicational code. As a trivial "lower bound", covert channel analysis shows us that any stateful shared component is capable of conveying information between its sharers [see Millen 1999]. More obviously, many boxes in box-and-line diagrams (such as in the *Capitalize* example of Allen and Garlan) have a clear communicational role. For example, the filters in a pipe-and-filter system are clearly communicational, since as well as transforming their input data, they forward the result to their output.

This does not mean that a distinction is not useful for purposes of convenience or modelling. It is precisely this convenience which is advocated by [Allen and Garlan 1997]. This is analogous with box-and-line diagrams: lines are merely very long and thin boxes, but we prefer to treat them differently for purposes of abstraction. We distinguish components based on what role (computation

or communication) is more significant when understanding the system at the depicted level of abstraction.

An important corollary of this is that we cannot make decisions about tools or implementation languages based on whether something "is a component" or "is a connector". During implementation, developers should avoid fixing a use-case in mind, for the sake of re-usability. However, the architectural significance of a piece of implementation clearly depends on its abstract role within a wider system, and hence on the context of its use. Consequently, we would ideally like to choose languages, tools and styles for connector implementation on a case-by-case basis from the full available range, just as we would when developing components.

This contrasts with the approach of UniCon [see Shaw et al. 1995], where introducing a new connector type involves extending the compiler [see Zelesnik 2000], and brings substantial constraints of programming language and style. In practice, it is quite likely that UniCon users would end up describing at least *some* communication abstractions as regular components, simply using the built-in connectors to wire them up. This avoids the inconvenience of extending the compiler, but is clearly contrary to UniCon's intended usage. While this does not undermine UniCon's usefulness as a practical system, it suggests that better solutions are possible, once the context-dependent nature of the component–connector distinction is understood.

3.2.7 Summary

The observations underlying the concept of connectors are insightful and uncontroversial. Communication abstractions are underrepresented in traditional programming practices, and better modularisation and separation of communicational concerns can provide great benefits to re-use. The distinction between *mechanism* and *agreement* provides a clearer idea of connectors than do distinctions of protocol, interface or state. Connectors provide mechanisms and demand agreements; agreements are the source of coupling, and adaptors are connectors which demand *non-complementary* agreements. [Tab. 4] lists a few examples of connectors and distinguishes them according to the various criteria described in this section.

3.3 Loose coupling

"Loosely coupled" is a term used frequently to describe some perceived good properties of systems. Unfortunately, this is a rather underspecified term which can be misleading. Here I will critically examine its use in both the coordination and service-oriented architecture literature.

Connector type	procedure call	remote procedure call	pipe	shared associative store	publish-subscribe network	2-way synchronisation barrier
Roles and parameters (static, *dynamic*)	caller component S callee component R entry point P in R	caller S callee *R* entry point P in R	writer W reader R	participants P_i	publishers P_i subscribers S_i	participants P_1, P_2
Mechanism names	S: invoke	S: invoke	W: write R: read	P_i: in, out	P_i: publish E S_i: subscribe	P_1, P_2: enter
Typical layered agreements	signature pre-/post-conditions approx duration wait/block behaviour	(as left)	data meaning additional behaviour	tuple structures tuple meanings referential/semantic constraints sequencing conditions	message structure meanings for standard headers/metadata e.g. topic	entry conditions
Example implementation technique(s)	stack-supported branch and return	stub/skeleton, marshalling to/from agreed network encoding, transmission by network datagram service	kernel-managed ring buffer	distributed hash table	network of store-and-forward broker nodes	lightweight threading and condition variables
Typical connector dependencies (immediate)	program counter increment shared registers shared memory (stack) branch instruction	local procedure call network datagram service	system call	procedure call network datagram service	network stream service	shared semaphores thread scheduler
Example instance(s)	(any local early-bound non-inlined procedure call)	Web Service calls e.g. in software update services	Unix pipe e.g. between `tar` and compression filter	Chord network	Usenet NNTP network (including user agents)	(e.g. any Barrier instance in a Java program)
Run-time representation or state	caller/callee instruction sequences	stub/skeleton code, file handle table, socket descriptor	kernel ring buffer, file descriptors	node storage, node forwarding tables, node proximity data	stored messages stored subscriptions overlay network topology	semaphore state
Transient invocation state	stack frame saved register values	(in dependencies only, i.e. in implementations of datagram service and procedure call)	(in dependencies only)	(in dependencies only)	(in dependencies only)	(in dependencies only)

Table 4: Examples of communication abstractions, highlighting distinctions between type, state and other attributes

In the coordination literature, loose coupling is claimed an advantage of most coordination models [see Carriero and Gelernter 1989; Diakov and Arbab 2004]. Intuitively, these coordination models are *loose* in the sense that there are few constraints on concurrent execution, and that the complexity (or at least the *syntactic* complexity) of the interface upon which each party has to agree—

typically a simple set of *in* and *out* operations—is small.

Meanwhile, "loose coupling" is also cited in the service-oriented architecture literature. [He 2003] explains this by contrasting "real" and "artificial" dependencies: real dependencies arise from functionality, such as a powered device requiring electricity, whereas artificial dependencies arise from arbitrary mismatches of integration details, such as the physical shape of plug accepted by a power outlet. Achieving loose coupling is the exercise of minimising artificial dependencies. He claims that service-oriented architectures achieve loose coupling by two techniques: the use of "small set of simple and ubiquitous interfaces... [where] only generic semantics are encoded", and the use of "descriptive messages" based on extensible schemas for communicating with services. He cites the REST architectural style [see Fielding 2000] as emblematic of this approach.

To evaluate these claims, we must revisit the origins, definitions and treatment of coupling, and the intuitions behind them.

3.3.1 A brief history of coupling

Coupling is a property of structured systems. [Stevens et al. 1974] provided the first definition of coupling in software systems, as "the measure of the strength of association established by a connection from one module to another".

The idea of "strength of association" captures two related empirical properties of software. Firstly, it includes the likelihood that that changes in one subsystem will require consequent changes in a disjoint (but coupled) subsystem. In this case, low coupling predicts extensibility and maintainability. Secondly, it correlates with difficulty of re-use: 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.

Two quantities of coupling can be measured: *global coupling*, which is an aggregate figure for the total strength of coupling within a modular system, and *local coupling*, which is the strength of the specific association between one component and its environment (or another specific component). Since we are concerned here with properties of individual components, I consider only the latter.

Coupling is measured in ad-hoc and language-specific ways. Stevens et al provided an ordinal scale of perceived severity of coupling. This was derived solely from experience, and defined in terms of the different modes of inter-module reference available in the procedural languages of the day. [Fenton and Melton 1990] added some rigour by grounding this in measurement theory, but the weighting of each mode of reference remained ad-hoc. Various measures have been defined on object-oriented models of software [see Chidamber and Kemerer 1994; Briand et al. 1996; Harrison et al. 1998], even in language-independent fashion [see Lanza and Ducasse 2002; Baroni et al. 2002; McQuillan and Power 2006].

However, so-called "language independent" measures rely on a unifying meta-model. It is comparatively straightforward to to devise a unifying meta-model for object-oriented languages, and hence gain a limited sense of language independence. However, to this author's knowledge, there are no measures that account for coupling which is not accompanied by explicit programmatic reference. Such coupling is common—for example, it occurs wherever logic concerning protocol implementation or string formatting conventions is spread across multiple communicating systems or modules.

As an example of the latter, consider a Unix pipeline of the form

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

which lists the names of all the environment variables defined in the current shell. The two programs interacting across the pipeline are coupled by the fact that every line output by `printenv` contains an equals sign. If `printenv` were changed, say to output a header line, then the pipeline's meaning would be changed, since the output would include the unwanted header line. This kind of coupling is similar to "stamp coupling" as described by Stevens, but clearly would not be captured by an analysis of programmatic reference.

3.3.2 Inevitability of coupling

Coupling is, fundamentally, a consequence of communication. I submit that it is not possible for two entities, whatever their kind, to communicate without at the same time being coupled *to some degree*. Additionally, if two entities do not communicate *through any path*, i.e. neither directly with each other nor indirectly through some intermediate entity, then necessarily they are outside each other's cones of influence, and so they cannot be not coupled [see 22].

Intuitively one expects communication to be *dynamic*, in that it happens during program execution. However, clearly, coupling can be static, in that we can observe associations and mismatches between source files without executing them. These might be caused by simple inter-module *references* in any source language, or alternatively perhaps by compile-time composition of templates in a language like C++. In these cases, it is not immediately apparent who is communicating with whom, or if indeed communication is occurring at all. Nevertheless, I claim that these observations of coupling are still fully captured by models of communication.

To understand this, consider the example of mismatched programmatic reference. If I change the name of some function or data structure member in one

[22] One could imagine an intermediate case where two entities are within each other's cones of influence, and exchange *information* in the sense of random events, but do not exchange *meaning* in that they have established no agreements on the meaning of each event.

source file, without updating the others to reflect this, the system will break at compile-time. This is simply because the compiler has detected that the respective modules' *communicational codes*—mappings from symbols to meanings, or *references* to *referents*—are incompatible. The name used in one module is no longer part of the "code" understood by the other modules. In less statically checked languages, such as Smalltalk, fewer errors are caught in the compiler: instead, they appear at run-time, once an actual dynamic occurrence of communication has failed [see 23].

3.4 Dealing with coupling

Many features of programming languages and software engineering practice are implicitly intended to reduce coupling. We split these into two categories: *minimisation* and *mitigation*. Minimisation techniques are designed to limit the actual extent of coupling, for example in reducing the complexity of the interface exposed by a module. Mitigation techniques are intended to reduce the practical harm caused by any coupling which cannot be reduced by minimisation.

3.4.1 Minimisation

Minimisation techniques are designed to prevent unnecessary coupling. Information hiding [see Parnas 1972] is the most obvious, and works by confining the extent of expressible inter-module agreements to interface definitions. The gains of this approach come from controlling the human tendency to unwittingly write needlessly specialised code: if knowledge of an implementation detail is not necessary, it will not be part of the interface, whereas if it eventually does prove necessary, the interface will be somehow revised to accommodate it.

An alternative approach comes from the observation that coupling may be dynamic as well as static. Consider how not only *classes* but individual *objects* may be coupled. The technique is therefore to reduce static coupling by pushing it into the dynamic realm. Late binding techniques (including virtual dispatch) take this approach. Negotiation in network protocols (for example character encoding negotiation in HTTP) is another example. These are sensible techniques since, by deferring decisions, these commitments may be made at a finer grain, both in

[23] We might more properly distinguish this mere "reference" from "communication": we could say that coupling is a consequence of *reference*, and that communication is simply a higher-layer process which makes use of reference. However, in the case of software, communication is a more intuitively transparent concept than that of reference. Moreover, in the case of software the distinction is rarely helpful, because it is easily affected by implementation details. For instance, in the above example, the fact that the Smalltalk compiler catches fewer errors than a more static-checking compiler is of no consequence to the abstract relationship between the two modules concerned. We therefore prefer to say that the modules "communicate" in both cases, whether or not the compiler implementation happens to "pre-compute" the attempted exchange of meaning.

space (by committing individual objects, rather than whole classes) and in time (since commitments hold only for a particular object's lifetime). Of course, they introduce run-time overhead.

Minimisation can only go so far. As we stated earlier, the ability to meaningfully communicate relies on some degree of coupling. Mitigation techniques are, therefore, essential complements to minimisation.

3.4.2 Localisation

We use the term "localisation" to refer to the many features of programming languages which enable definitions to be made in a single place but referenced from many others. Examples include symbolic constants, functions, macros, and aspects. All of these features enable the localisation of definitions which would otherwise be repeated in-line.

Localisation is most obviously a mitigation technique, since it reduces the practical cost of embedding assumptions about the outside (by keeping such assumptions localised). However, since it may reduce the number of distinct external references in source code, it may also reduce the coupling observed by various measures, and therefore may also be considered a minimisation technique.

3.4.3 Standardisation

Standardisation is a popular technique for mitigation of coupling, by encouraging the adoption of some standard *global* agreements. When interface details are standardised across some modular boundary, coupling remains present and measurable—changing the details of one module would still require changes to the other. However, the coupling is less problematic in practice, because all candidates for interoperation are also written against the same standard. Maintenance is easier because the standard is stable. Comprehensibility is easier because developers are more likely to be familiar with the standard.

There are countless well-known examples of successful standards: the ASCII character set, Unix tools, the C standard library, the Internet protocol, HTML, and many more. However, there are functional limitations to any standard—for example, the inability of ASCII to encode certain non-Western or punctuation characters. Complex standards may be impractical to implement, and may consequently fail to gain wide adoption (as with some of the OMG's CORBA standards [see Henning 2006] or C++ export templates [see Sutter and Plum 2003]). The fragility of standards is a problem, such as when non-conformant implementations are widely deployed (either unwittingly or maliciously) [see Ts'o 1997; Zelnick 1998]. Nonconformance requires workarounds, with consequent maintenance overheads. (These workarounds can be seen as adaptation, and improved adaptation techniques would reduce these maintenance overheads.)

Even where a standard is successfully made and deployed, innovation will eventually cause the standard to be superseded. This inherent expense and fragility makes standardisation suited only to technologies which demand wide deployment but expect only infrequent innovation. There are many such domains, including network protocols, data encodings and programming languages. However, there are innumerably more which do not fit these criteria. Many software components perform a function that is unique to them; they are often continually evolving. Some of these changes invariably necessitate changed interface agreements. If we are to maximise the potential for re-use in these scenarios, some other approach is necessary.

The main value of standards is therefore in preventing *unnecessarily many* competing conventions. By contrast, there will always be reasons why some number of differing conventions are necessary or unavoidable: functional differences, local customisation, parallel development, innovation, experimentation, competition, and so on. In order to compose and re-use artifacts written against different conventions, a complementary technique is necessary—this, by definition, is adaptation.

3.4.4 A more general theory of coupling

So far we have seen no way of evaluating or comparing the claimed "loose coupling" found in the surveyed work. Here I outline an intuition about coupling which enables us to do so.

We have already seen the `printenv` pipeline example of coupling not captured by existing measurement techniques. Consider, similarly, 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 clearly be understood by both parties in order for them to communicate, and is therefore a source of coupling.

Clearly, 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. In other words, the coding scheme upon which communication depends is *less complex*. We can therefore evaluate claims of "loose coupling" based on whether they result in less complex communicational codes between units of software.

3.4.5 Evaluating the claims

We undertook this exploration of coupling in order to evaluate the claims that coordination models, such as that of Linda, and service-oriented architectural styles, such as the example of REST used by He, lead to loosely-coupled systems.

In the case of Linda, we see that the complexity of the interface upon which party must agree with the tuple space is very small—it is simply the set of operations such as *read*, *in* and *out*. This is the source of the perceived elegance of Linda. However, in order for a Linda process to communicate *with another process*, there must be agreements over and above what each process agreed with the tuple space. There are two obvious kinds of agreement: tuple structures, i.e. the number and types of fields in each tuple; and protocol, i.e. the ordering conventions by which different processes read tuples and deposit new ones.

Clearly, the latter two kinds of coupling are of great practical importance when re-using or evolving elements of a Linda-based system. Unfortunately, since Linda provides no way to explicitly define the conventions in a well-modularised, localised fashion, the coupling is not well mitigated. More subtly, since the communication primitives of Linda are so simple, there are innumerably many ways in which independently developed Linda processes might have chosen to encode communication details (such as tuple structures or protocol) of equivalent meaning. This makes the likely extent of mismatch very great. To use He's words, it leads to unnecessary *artificial dependencies*. Consider a Linda process that wishes to pass a list of tuples to another: aside from mismatches of individual tuple structure, it might encode the list in a one-at-a-time fashion, by publishing tuples with a fixed "list ID" field value, or in a many-at-a-time fashion by publishing tuples with contiguous sequence numbers.

Meanwhile, service-oriented architectures appear to suffer similar problems. Their reliance on a small, simple, "ubiquitous" interface with only "generic" semantics, means that little semantic detail is present in a request. This approach mirrors the Unix philosophy of "everything is a file". Whether our objects are services, files or resources, this unification-based approach suffers the same drawbacks: semantic detail becomes implicit within component implementations, rather than explicit within their interfaces. Little static checking can be performed, owing to this lack of semantic detail. In practice most objects will implement some ill-defined subset of the unified interface, discoverable only at run-time. 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 (the approach required by REST), 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.

Service-oriented architectures do improve on both Linda and Unix-like mod-

els, by including metadata with a request. This allows a service to determine which particular version of itself is required by the client, and to service the request appropriately. This is precisely the negotiation-based coupling minimisation technique described earlier (see [Section 3.4.1]). Service-oriented architectures therefore suffer less coupling, in like-for-like terms, than either Linda or Unix-like schemes. However, as always, this negotiation technique has its limits: only interface variations known to the service may be catered for [see 24]. A more general, open and semantically-aware interface negotiation technique remains a current research goal [see Canal 2004].

The problems shared by both Linda and REST are a consequence of choosing a simple set of communicational primitives. Clearly, at least, most higher-level application code should not be written directly targetting these "send-receive" style interfaces. [Gorlatch 2004] gives a comparable critique of send-receive in the domain of parallel programming. Of course, there is a danger that a larger and more semantically rich set of primitives would destroy other benefits related these systems' simplicity, such as ease of implementation. Similarly, we saw earlier that primitives may be over-specific—for example Unix's separate treatment of shared-memory objects and files, or Java's unhelpful distinction between static state and objects. This introduces arbitrary distinctions, causing unnecessary coupling. Finding optimal points along this space is a hard problem, and those points depend greatly on the required mix of other benefits. Accordingly, I do not intend to dismiss the benefits of coordination models, nor of service-oriented architectures. However, the term "loose coupling" does require careful qualification.

4 Modularisation strategies

Some adaptation techniques, such as the several Java-language transformation engines surveyed earlier, simply advocate their own use when adaptation is necessary—that is, if there happens to be a mismatch. Others propose that systems are habitually written using a particular division of languages or styles. In the survey we saw many examples of systems employing *configuration languages*. These languages span a spectrum from architecture description languages down to linking languages. A final design issue to examine, therefore, is as follows. If we accept the value of some split between configuration languages and conventional programming languages, how should the entire system ideally be modularised?

[24] Giving credit to REST, its application to such a wide domain as service-oriented architectures goes far beyond its original aims, which were simply to capture the architectural style of the World-Wide Web as a non-computational hypertext storage and retrieval system. In this context, its small set of storage-oriented operations are a sound design. My discussion of it here is a critique of the over-application of its design by third parties, rather than of its intrinsic merits.

Which *concerns* should be separated by this split? Here I will summarise the separations encouraged by configuration languages surveyed in [Section 2].

Stable from unstable This separation is mentioned explicitly as a design goal of Piccola [see Nierstrasz and Achermann 2000], and implicitly informs the design other scripting and orchestration languages. It is intuitively reasonable, not only because it encourages the localisation and modularisation of unstable parts of a system, but because specialised languages are likely to do a better job of making it efficient to develop and modify unstable code. However, it doesn't distinguish the many kinds of changes with respect to which code might be unstable: perhaps different re-use contexts, or perhaps rapid prototyping within the same project.

Proactive from reactive This separation is again exploited by scripting and orchestration languages, and also characterises service-oriented computing. It also answers adaptation needs in the common re-use case where the re-used artifacts are existing non-proactive services, and the essentially novel artifact is a script which adapts and composes them. However, we note that not all cases are well-served by this separation, since both proactive and reactive components may wish to be re-used—for instance in Garlan's example of re-using event loops [see Garlan et al. 1995].

Integration from functionality This separation captures the purest ideal of software re-use: that any implementation of the desired functionality might be re-used without concern for *how* it is implemented. Nimble [see Purtilo and Atlee 1991], Polygen [see Callahan 1993] and Flexible Packaging [see DeLine 2001] all target this separation. Unfortunately, realising pure ideals is a hard problem, and each of these systems puts constraints on how functionality can be written: it must be written against either a procedural abstraction or, in the case of Flexible Packaging, an asynchronous channel abstraction.

Structure from content This is a weaker version of the previous separation, and is adopted by linking languages such as Knit [see Reid et al. 2000] and configuration languages such as Darwin [see Magee et al. 1995]. Although not addressing traditional adaptation problems, such as mismatches of data encoding or protocol, explicit structure avoids the maintenance overheads associated with such error-prone conventions as name-matching. Network-based coordination models, such as that of Reo [see Arbab and Mavaddat 2002], also provide this structural separation, while additionally being capable of expressing protocol adaptation.

Computation from communication This separation is an often-cited goal in many of the research areas we have surveyed, including coordination [see

Papadopoulos and Arbab 1998] and software architecture [see Shaw 1994]. It relates to precisely the observations which motivated the concept of connectors: communication abstractions are comparatively neglected (by contrast with computational abstractions such as data structures and algorithms), and conventional programming languages don't adequately localise details of communication. Unfortunately, this separation is not completely possible: clearly even a maximally computation-oriented component must somehow express or imply the communication of its results to its environment. A simplistic answer is to allow only very primitive expressions of communication from within computational components, but this leads to the problems of insufficient abstraction mentioned in [Section 3.3]. Other separations, particularly that between functionality and integration, may provide better ways of framing the problem.

Language-specific concerns Binary Component Adaptation [see Keller and Holzle 1998], in its ability to augment classes and alter typing information, separates units of implementation from linguistic units of typing or instantiation (the latter being, specifically, Java classes). This highlights the potential for more complex type systems to capture additional classes of mismatch, as more recently echoed by [Bracha 2004]. This is perhaps a more specific version of the concerns separated by Swig [see Beazley 1996], which separates more abstract functionality from the various language-specific ways of reifying those functions as language constructs.

5 Summary

I have reviewed many adaptation techniques, and many aspects of the design of adaptation techniques. By examining the claims and intentions behind various designs, I hope to have provided a clearer understanding of various considerations and trade-offs. We have seen a very wide design space, and reviewed some ideas which help us decide what constitutes a *good* point in that space with respect to particular requirements. We have also seen how the concept of *coupling* is a recurrent and a unifying one when making such decisions.

The design of configuration languages, and the appropriate separations of concerns, is also of prime importance. Ideally we should like an approach separates all of the concerns just described, embodying a component model which encourages minimal coupling, and whose adaptation features effectively mitigate whatever coupling is unavoidable. For this, we must look to future work.

Acknowledgement

The author would like to thank his supervisor Dr David Greaves, and several other people who have made helpful comments and suggestions: Jon Crowcroft,

Theodore Hong, Andrew Moore, Derek Murray, Henry Robinson, Mark Williamson and the anonymous reviewers.

References

[Achermann and Nierstrasz 2001] Achermann, F., Nierstrasz, O.: "Applications = components + scripts"; Software Architectures and Component Technology; 261–292; Kluwer, 2001.

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

[Arbab 1998] Arbab, F.: "What do you mean, coordination"; Bulletin of the Dutch Association for Theoretical Computer Science, NVTI; 1122 (1998).

[Arbab and Mavaddat 2002] Arbab, F., Mavaddat, F.: "Coordination through channel composition"; Proc. Coordination; 21–38; 2002.

[Arbab and Rutten 2003] Arbab, F., Rutten, J. J. M. M.: "A coinductive calculus of component connectors"; Proceedings of 16th International Workshop on Algebraic Development Techniques (WADT 2002); 35–56; Springer-Verlag, 2003.

[Assmann et al. 2000] Assmann, U., Genssler, T., Bar, H.: "Meta-programming grey-box connectors"; Proceedings of 33rd International Conference on Technology of Object-Oriented Languages (TOOLS 33); 300–311; 2000.

[Barbosa and Barbosa 2004] Barbosa, M., Barbosa, L.: "Specifying software connectors"; 1st International Colloquium on Theorectical Aspects of Computing (ICTAC '04); 53–68; 2004.

[Barham et al. 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: "Xen and the art of virtualization"; SIGOPS Oper. Syst. Rev.; 37 (2003), 5, 164–177.

[Baroni et al. 2002] Baroni, A. L., Braz, S., Abreu, O. B. E., Portugal, N. L.: "Using OCL to formalize object-oriented design metrics definitions"; Proceedings of 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering; Springer-Verlag, 2002.

[Beazley 1996] Beazley, D.: "Swig: An easy to use tool for integrating scripting languages with C and C++"; Proceedings of the 4th USENIX Tcl/Tk Workshop; 129–139; 1996.

[Birrell and Nelson 1984] Birrell, A., Nelson, B.: "Implementing remote procedure calls"; ACM Transactions on Computer Systems (TOCS); 2 (1984), 39–59.

[Bosch 1999] Bosch, J.: "Superimposition: a component adaptation technique"; Information and Software Technology; 41 (1999), 257–273.

[Bracciali et al. 2005] Bracciali, A., Brogi, A., Canal, C.: "A formal approach to component adaptation"; The Journal of Systems & Software; 74 (2005), 45–54.

[Bracha 2004] Bracha, G.: "Pluggable type systems"; OOPSLA Workshop on Revival of Dynamic Languages; 2004.

[Bracha et al. 1993] Bracha, G., Clark, C., Lindstrom, G., Orr, D.: "Module management as a system service"; OOPSLA Workshop on Object-oriented Reflection and Metalevel Architectures; 1993.

[Bracha and Cook 1990] Bracha, G., Cook, W.: "Mixin-based inheritance"; ECOOP/OOPSLA '90 Proceedings; 303–311; 1990.

[Briand et al. 1996] Briand, L., Morasca, S., Basili, V.: "Property-based software engineering measurement"; IEEE Transactions on Software Engineering; 22 (1996), 68–86.

[Callahan 1993] Callahan, J.: Software packaging; Ph.D. thesis; University of Maryland (1993).

[Callahan and Purtilo 1991] Callahan, J., Purtilo, J.: "A packaging system for heterogeneous execution environments"; IEEE Transactions on Software Engineering; 17 (1991), 626–635.

[Canal 2004] Canal, C.: “On the dynamic adaptation of component behaviour”; First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT ’04); 2004.

[Carriero and Gelernter 1989] Carriero, N., Gelernter, D.: “Linda in context”; Communications of the ACM; 32 (1989), 444–458.

[Chakravarty et al. 2002] Chakravarty, M., Finne, S., Henderson, F., Kowalczyk, M., Leijen, D., Marlow, S., Meijer, E., Panne, S.: “The Haskell 98 foreign function interface 1.0: an addendum to the Haskell 98 report”; (2002).

[Channabasavaiah et al. 2003] Channabasavaiah, K., Holley, K., Tuggle, E.: “Migrating to a service-oriented architecture”; IBM DeveloperWorks; (2003); available at http://www.ibm.com/developerworks/library/ws-migratesoa/, retrieved 2008-08-26.

[Chiba 2000] Chiba, S.: “Load-time structural reflection in java”; ECOOP 2000 Proceedings; 2000.

[Chidamber and Kemerer 1994] Chidamber, S., Kemerer, C.: “A metrics suite for object oriented design”; IEEE Transactions on Software Engineering; 20 (1994), 476–493.

[Cohen et al. 1998] Cohen, G., Chase, J., Kaminsky, D.: “Automatic program transformation with JOIE”; Proceedings of the USENIX Annual Technical Conference; 14; 1998.

[Councill and Heineman 2001] Councill, B., Heineman, G.: “Definition of a software component and its elements”; Component-based software engineering: putting the pieces together; 5–19; Addison Wesley, 2001.

[DeLine 2001] DeLine, R.: “Avoiding packaging mismatch with flexible packaging”; IEEE Transactions on Software Engineering; 27 (2001), 124–143.

[DeRemer and Kron 1975] DeRemer, F., Kron, H.: “Programming-in-the large versus programming-in-the-small”; Proceedings of the International Conference on Reliable Software; 114–121; 1975.

[Devine et al. 2002] Devine, S., Bugnion, E., Rosenblum, M.: “Virtualization system including a virtual machine monitor for a computer with a segmented architecture”; United States Patent 6397242 (2002).

[Diakov and Arbab 2004] Diakov, N., Arbab, F.: “Compositional construction of web services using Reo”; Technical Report SEN-R0406; CWI; Amsterdam (2004).

[Eisenbach et al. 2007] Eisenbach, S., Sadler, C., Wong, D.: “Component adaptation in contemporary execution environments”; Proceedings of 7th IFIP International Conference on Distributed Applications and Interoperable Systems; Springer-Verlag, 2007.

[Eterovic et al. 2004] Eterovic, Y., Murillo, J., Palma, K.: “Managing components adaptation using aspect oriented techniques”; First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT ’04); 2004.

[Fenton and Melton 1990] Fenton, N., Melton, A.: “Deriving structurally based software measures”; Journal of Systems and Software; 12 (1990), 177–187.

[Fielding 2000] Fielding, R. T.: Architectural styles and the design of network-based software architectures; Ph.D. thesis; University of California, Irvine (2000).

[Filman and Friedman 2000] Filman, R., Friedman, D.: “Aspect-oriented programming is quantification and obliviousness”; OOPSLA Workshop on Advanced Separation of Concerns; 2000.

[Fowler 2004] Fowler, M.: “Inversion of control containers and the dependency injection pattern”; Web document (2004); available at http://martinfowler.com/articles/injection.html, retrieved 2008-08-26.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software; Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[Garlan et al. 1995] Garlan, D., Allen, R., Ockerbloom, J.: “Architectural mismatch or why it’s hard to build systems out of existing parts”; Proceedings of the 17th International Conference on Software Engineering; 179–185; 1995.

[Garlan and Shaw 1994] Garlan, D., Shaw, M.: “An introduction to software architecture”; Technical Report CMU-CS-94-166; School of Computer Science, Carnegie Mellon University (1994).

[Gelernter and Carriero 1992] Gelernter, D., Carriero, N.: "Coordination languages and their significance"; Communications of the ACM; 35 (1992), 97–107.

[Gorlatch 2004] Gorlatch, S.: "Send-receive considered harmful: Myths and realities of message passing"; ACM Transactions on Programming Languages and Systems (TOPLAS); 26 (2004), 47–56.

[Haack et al. 2002] Haack, C., Howard, B., Stoughton, A., Wells, J.: "Fully automatic adaptation of software components based on semantic specifications"; Proc. 9th Int'l Conf. Algebraic Methodology & Softw. Tech.; 2002.

[Harrison et al. 1998] Harrison, R., Counsell, S., Nithi, R.: "Coupling metrics for object-oriented design"; Proceedings of Fifth International Software Metrics Symposium; 150–157; 1998.

[Harrison and Ossher 1993] Harrison, W., Ossher, H.: "Subject-oriented programming: a critique of pure objects"; ACM SIGPLAN Notices; 28 (1993), 411–428.

[He 2003] He, H.: "What is service-oriented architecture"; (2003); available at http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html, retrieved 2008-08-26.

[Henning 2006] Henning, M.: "The rise and fall of CORBA"; ACM Queue; 4 (2006), 28–34.

[Järvi et al. 2007] Järvi, J., Marcus, M., Smith, J.: "Library composition and adaptation using C++ concepts"; Proceedings of the 6th International Conference on Generative Programming and Component Engineering; 73–82; 2007.

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

[Kepser 2004] Kepser, S.: "A simple proof for the Turing-completeness of XSLT and XQuery"; Proceedings of Extreme Markup Languages 2004; 2004.

[Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: "An overview of AspectJ"; ECOOP 2001; Springer-Verlag, 2001.

[Kiczales et al. 1997] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: "Aspect-oriented programming"; ECOOP 1997; Springer-Verlag, 1997.

[Kniesel et al. 2001] Kniesel, G., Costanza, P., Austermann, M.: "JMangler: a framework for load-time transformation of Java classfiles"; Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation; 98–108; 2001.

[Kouici et al. 2005] Kouici, N., Conan, D., Bernard, G.: "An experience in adaptation in the context of mobile computing"; Second International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT '05); 2005.

[Lanza and Ducasse 2002] Lanza, M., Ducasse, S.: "Beyond language independent object-oriented metrics: Model independent metrics"; Proceedings of 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering; 2002.

[LeVasseur et al. 2004] LeVasseur, J., Uhlig, V., Stoess, J., Götz, S.: "Unmodified device driver reuse and improved system dependability via virtual machines"; Proceedings of the 6th Symposium on Operating Systems Design & Implementation; USENIX Association, San Francisco, CA, 2004.

[Liang 1999] Liang, S.: The Java Native Interface: Programmer's Guide and Specification; Addison-Wesley Professional, 1999.

[Magee et al. 1995] Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: "Specifying distributed software architectures"; Proceedings of the 5th European Software Engineering Conference; 137–153; 1995.

[McDirmid et al. 2001] McDirmid, S., Flatt, M., Hsieh, W.: "Jiazzi: new-age components for old-fasioned Java"; Proceedings OOPSLA 2001; 211–222; 2001.

[McIlroy 1969] McIlroy, M.: "Mass-produced software components"; Proceedings of NATO Conference on Software Engineering; 88–98; 1969.

[McQuillan and Power 2006] McQuillan, J., Power, J.: "Towards re-usable metric definitions at the meta-level"; PhD Workshop of ECOOP 2006; 2006.

[Mehta et al. 2000] Mehta, N., Medvidovic, N., Phadke, S.: "Towards a taxonomy of software connectors"; Proceedings of the 22nd International Conference on Software Engineering; 178–187; 2000.

[Meijer 2002] Meijer, E.: "Technical overview of the common language runtime"; language; 29 (2002), 7.

[Millen 1999] Millen, J.: “20 years of covert channel modeling and analysis”; Proceedings of the 1999 IEEE Symposium on Security and Privacy; 113–114; 1999.
[Misra and Cook 2006] Misra, J., Cook, W.: “Computation orchestration: A basis for wide-area computing”; Journal of Software and Systems Modeling; 6 (2006), 83–110.
[Nierstrasz and Achermann 2000] Nierstrasz, O., Achermann, F.: “Separation of concerns through unification of concepts”; ECOOP 2000 Workshop on Aspects & Dimensions of Concerns; 2000.
[Ossher et al. 1995] Ossher, H., Kaplan, M., Harrison, W., Katz, A., Kruskal, V.: “Subject-oriented composition rules”; Proceedings of OOPSLA 1995; 235–250; 1995.
[Ousterhout 1998] Ousterhout, J.: “Scripting: higher level programming for the 21st century”; Computer; 31 (1998), 23–30.
[Owens 2007] Owens, S.: Compile time information in software components; Ph.D. thesis; University of Utah (2007).
[Padioleau et al. 2008] Padioleau, Y., Lawall, J., Hansen, R. R., Muller, G.: “Documenting and automating collateral evolutions in Linux device drivers”; Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008; 247–260; ACM, 2008.
[Papadopoulos and Arbab 1998] Papadopoulos, G., Arbab, F.: “Coordination models and languages”; Technical Report SEN-R9834; CWI; Amsterdam (1998).
[Papazoglou 2003] Papazoglou, M.: “Service-oriented computing: concepts, characteristics and directions”; Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003); 3–12; 2003.
[Parnas 1972] Parnas, D.: “On the criteria to be used in decomposing systems into modules”; Communications of the ACM; 15 (1972), 1053–1058.
[Passerone et al. 2002] Passerone, R., de Alfaro, L., Henzinger, T., Sangiovanni-Vincentelli, A.: “Convertibility verification and converter synthesis: Two faces of the same coin”; Proceedings of the International Conference on Computer-Aided Design 2002; 2002.
[Peltz 2003] Peltz, C.: “Web services orchestration and choreography”; Computer; 36 (2003), 46–52.
[Pilhofer 1999] Pilhofer, F.: Design and Implementation of the Portable Object Adapter; Sulimma, Frankfurt, 1999.
[Pisello 2006] Pisello, T.: “Is there real business value behind the hype of SOA?”; IDG Computerworld; (2006).
[Purtilo and Atlee 1991] Purtilo, J., Atlee, J.: “Module reuse by interface adaptation”; Software - Practice and Experience; 21 (1991), 539–556.
[Reid et al. 2000] Reid, A., Flatt, M., Stoller, L., Lepreau, J., Eide, E.: “Knit: Component composition for systems software”; Proc. of the 4th Operating Systems Design and Implementation (OSDI); 347–360; 2000.
[Reussner 2003] Reussner, R.: “Automatic component protocol adaptation with the CoConut/J tool suite”; Future Generation Computer Systems; 19 (2003), 627–639.
[Rine et al. 1999] Rine, D., Nada, N., Jaber, K.: “Using adapters to reduce interaction complexity in reusable component-based software development”; Proceedings of the 1999 Symposium on Software Reusability; 37–43; 1999.
[Seeley 1990] Seeley, D.: “Shared libraries as objects”; USENIX 1990 Summer Conference Proceedings; 25–37; 1990.
[Serra et al. 2000] Serra, A., Navarro, N., Cortes, T.: “DITools: application-level support for dynamic extension and flexible composition”; ATEC ’00: Proceedings of the USENIX Annual Technical Conference; 19–19; USENIX Association, Berkeley, CA, USA, 2000.
[Shannon and Weaver 1949] Shannon, C., Weaver, W.: A Mathematical Theory of Communication; University of Illinois Press, 1949.
[Shaw 1994] Shaw, M.: “Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status”; Technical Report CMU/SEI-94-TR-002; Carnegie Mellon University (1994).
[Shaw 1995] Shaw, M.: “Architectural issues in software reuse: It’s not just the functionality, it’s the packaging”; Proc. IEEE Symposium on Software Reusability; 1995.

[Shaw et al. 1995] Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., Zelesnik, G.: "Abstractions for software architecture and tools to support them"; IEEE Transactions on Software Engineering; 21 (1995), 314–335.

[Stevens et al. 1974] Stevens, W., Myers, G., Constantine, L.: "Structured design"; IBM Journal of Research and Development; 13 (1974), 115.

[Sutter and Plum 2003] Sutter, H., Plum, T.: "Why we can't afford export"; ISO C++ committee paper (2003).

[Szyperski 1998] Szyperski, C.: Component Oriented Programming; Springer, 1998.

[Ts'o 1997] Ts'o, T.: "Microsoft "embraces and extends" Kerberos v5"; Usenix ;login:; Windows NT Special Issue (1997).

[Waldo and Clemsford 1998] Waldo, J., Clemsford, M.: "Remote procedure calls and Java Remote Method Invocation"; IEEE Concurrency; 6 (1998), 5–7.

[Wegner 1996] Wegner, P.: "Coordination as constrained interaction (extended abstract)"; Proceedings of the First International Conference on Coordination Languages and Models; 28–33; 1996.

[Wegner 1997] Wegner, P.: "Why interaction is more powerful than algorithms"; Communications of the ACM; 40 (1997), 80–91.

[Whitaker et al. 2004] Whitaker, A., Cox, R. S., Shaw, M., Grible, S. D.: "Constructing services with interposable virtual hardware"; Proceedings of the Symposium on Networked Systems Design and Implementation; 13–13; USENIX Association, San Francisco, California, 2004.

[Yellin and Strom 1997] Yellin, D., Strom, R.: "Protocol specifications and component adaptors"; ACM Transactions on Programming Languages and Systems; 19 (1997), 292–333.

[Zelesnik 2000] Zelesnik, G.: "Adding support for connector abstractions in the UniCon compiler"; Web document (2000); available at http://www.cs.cmu.edu/%7eUniCon/adding-connectors/expert-creation.html, retrieved 2008-08-26.

[Zelnick 1998] Zelnick, N.: "Nifty technology and nonconformance: the web in crisis"; Computer; 31 (1998), 115–116.