Appeared in the Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001,
Toronto, Ontario, May 2001.

Aspect Weaving as Component Knitting: Separating Concerns with Knit

Eric Eide Alastair Reid Matthew Flatt Jay Lepreau

University of Utah, School of Computing

{eeide,reid,mflatt,lepreau}@cs.utah.edu http://www.cs.utah.edu/flux/

Abstract

Knit is a new component specification and linking language. It was initially designed for low-level systems software, which requires especially flexible components with especially well-defined interfaces. For example, threads and virtual memory are typically implemented by components within the system, instead of being supplied by some execution environment. Consequently, components used to construct the system must expose interactions with threads and memory. The component composition tool must then check the resulting system for correctness, and weave the components together to achieve reasonable performance.

Component composition with Knit thus acts like aspect weaving: component interfaces determine the "join points" for weaving, while components (some of which may be automatically generated) implement aspects. Knit is not limited to the construction of low-level software, and to the degree that a set of components exposes fine-grained relationships, Knit provides the benefits of aspect-oriented programming within its component model.

1 Introduction

To ensure the flexibility and usability of components, a component model must assign well-specified interfaces to each component, so that programmers know how to use or replace them. The mechanism for linking components must give the programmer broad control over the way that components are combined, allowing the programmer to replace any component with another having the same interface, or even to use a particular component multiple times in the same system, each time with a different importing context.

This research was largely supported by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, under agreement numbers F30602–99–1–0503 and F33615–00–C–1696. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

US Mail contact address: University of Utah, School of Computing, 50 S. Central Campus Drive, Room 3190, SLC, UT 84112–9205.

In addition, to scale component programming to large and complex sets of components, the model must also address *cross-cutting* dependencies among components. These kinds of relationships occur naturally when building reusable software or component kits. For example, consider the problems faced by the designer of a component kit for operating systems development. At the system level, services such as virtual memory and threads, which are given *a priori* in the execution environment for applications, must be explicitly implemented by components. This in turn means that connections to these basic services should be made explicit: e.g., that every component that requires threads be designed so that it has an explicit dependency on the threading component. As connections become more and more exposed, the relationships between components become more and more complex, and often circular. The components cannot be organized into discrete layers, and the soundness of each connection involves dependencies that may span the entire system.

The need to define, manage, and check dependencies, despite their propagation throughout the system, was our motivation to create *Knit* [11], a new component specification and linking language that is suited to complex programming situations such as those encountered by systems code, middleware, and other complex and reusable software. Knit supports "manifest aspects" that involve the insertion of code or components into the system, as well as "non-manifest aspects" that involve reasoning about the correctness of component compositions. Knit provides configure-time flexibility: Knit performs aspect weaving and checking at system configure-time, to allow for maximum performance and assurance at system run-time. Because we targeted Knit to the needs of systems and middleware code, Knit works on C language components.

Knit is a work in progress. In the following sections we summarize the current language and tools (Section 2) and show how "aspect weaving" and "component knitting" are related (Section 3). We then describe planned future improvements to Knit to better support aspect-oriented programming (Section 4).

```
#include <stdio.h>
int main(int argc, char** argv) {
    printf("Hello, world!\n");
    return 0;
}
```

(a) The file `hello.c`, the canonical “Hello, world!” program.

```
unit Hello = {
    imports [ io: {printf} ];
    exports [ main: {main} ];
    depends { main needs io; };
    files { "hello.c" };
}
```

(b) The file `hello.unit`, the unit definition for “Hello, world!”

Figure 1: Defining an atomic unit with Knit.

2 Knit

The Knit component specification and linking language is based on *units* [5, 6], a model of components in the spirit of the Modula-3 [8] and Mesa [10] module systems. A unit is a component or model definition, either *atomic* (with a C implementation) or *compound* (composed of other units).

2.1 Building Blocks: Atomic Units

An *atomic unit* can be thought of as a module with three basic parts:

1. A set of *imports*, which name the functions and variables that must be supplied to the unit.
2. A set of *exports*, which name the function and variables that are defined by the unit and are made available to other units.
3. A set of top-level C declarations and definitions, which must include a definition for each exported name, and which may use any of the imported names as required. Defined names that are not exported are inaccessible from outside the unit. (The Knit compiler ensures that this is so.)

Figure 1(a) shows the C code for the canonical “Hello, world!” program, and Figure 1(b) shows the corresponding Knit unit definition. This simple unit, called `Hello`, imports a single function `printf` and exports a single function `main`. Functions and variables are imported and exported from units in groups called *bundles*. In the `Hello` unit, the `printf` function is imported as a member of the bundle called `io`, and the `main` function is exported as a member of the bundle called `main`. Bundles are critical for defining units with many imported or exported functions.

The unit does not contain the actual C definition of the `main` function; instead, the unit refers to the file `hello.c` in which the function is defined. This feature of Knit allows one to use Knit generally without modifying existing C code. The separation between a unit’s definition (in a ‘`.unit`’ file) and implementation (one or more ‘`.c`’ files) allows for other kinds of flexibility as well. For instance, one can define units at different levels of granularity: one can group several files into one unit for ease of specification, or one can make a separate unit for each file, to maximize configurability.

In addition to **imports**, **exports**, and **files**, the `Hello` unit definition contains a **depends** specification. This line provides an initialization constraint for the unit. In particular, the specification “`main needs io`” says that the functions in the `main` bundle call the functions that are imported from the `io` bundle. Knit uses this information in order to provide *automatic scheduling of unit initialization and finalization*, thus ensuring that all the unit instances within a compound unit instance will initialized and finalized in a correct order.

2.2 Linking: Compound Units

A set of units can be linked together to form a *compound unit*. The task of linking a set of units into a compound unit is to match the imports of each unit either with the exports of another unit or with the imports of the compound unit (to be linked to the exports of some other unit at a later time). Optionally, the exports of each unit can be propagated as exports of the compound unit. The result of composing units is a new unit, which is available for further linking.

Figure 2 shows how a compound unit can be defined by combining other units. (The units within a compound unit can be either atomic or compound, but in this example, each of the internal units is atomic.) The first unit, `Main`, imports two bundles called `io` and `msg`. These bundle each contain a single function, `printf` and `message` respectively, as defined by the appropriate **bundletype** definitions.

The second unit, `NWK`, imports nothing but exports a `msg` bundle, meaning that the unit exports a function called `message`. However, the code in Figure 2(b), which implements the `NWK` unit, does not define a function called `message`. Knit allows us to “fit the implementation to the interface,” however, through the use of a

```
#include <stdio.h>
const char *message(void);
int main(int argc, char** argv) {
    printf("%s", message());
    return 0;
}
```

(a) The file `main.c`, implementing unit `Main`.

```
const char *not_worth_knowing(void) {
    return
        ("A language that doesn't affect the "
         "way you think about programming is "
         "not worth knowing.\n");
}
```

(b) The file `nwk.c`, implementing unit `NWK`.

```
// Define shared bundle types.
bundletype IO_T   = { printf }
bundletype Msg_T  = { message }
bundletype Main_T = { main }

unit Main = {
    imports [ io: IO_T,
              msg: Msg_T ];
    exports [ main: Main_T ];
    depends { main needs (io+msg); };
    files { "main.c" };
}

unit NWK = {
    imports [];
    exports [ msg: Msg_T ];
    depends { msg needs {}; };
    files { "nwk.c" };
    rename {
        msg.message to not_worth_knowing;
    };
}

unit Program = {
    imports [ io: IO_T ];
    exports [ main: Main_T ];
    link {
        // [exports] <- Unit <- [imports];
        [msg]  <- NWK  <- [];
        [main] <- Main <- [io, msg];
    };
}
```

(c) The Knit unit definitions.

Figure 2: Defining a compound unit.

rename declaration. The declaration in `NWK` says that the `message` member of the `msg` bundle should be renamed, or implemented by, the C function `not_worth_knowing`. Renaming declarations are essential in Knit, to match existing code to "standardized" component interfaces.

Finally, the `Program` compound unit is defined by composing an instance of `Main` with an instance of `NWK`. On the first line of the **link** section, the exported bundle from `NWK` is called `msg`. On the second line, this bundle is listed as an import to the `Main` unit. The `io` bundle comes from the `Program` unit's list of imports; similarly, the `main` bundle that is implemented by the `Main` unit is propagated to be an export from the compound unit. If Knit is told to compile the `Program` unit, then the functions and variables imported by `Program` will be resolved by definitions from the environment outside Knit's world, e.g., the standard C library. Similarly, the exports from `Program` will be made available to the environment.

2.3 Constraints

Knit provides a flexible model for component specification and linking, thus making it easy to compose software units in myriad ways. But even if every single link in a system is "correct" according to local constraints such as type safety, the system *as a whole* may be incorrect because it does not meet certain global constraints.

Consider the task of building an operating system from components. Systems programmers are often concerned about the *execution environment* of a piece of code, and a simple but important aspect of an execution environment is the distinction that most operating systems make between *top-half* code — code called by a user process via system calls — and *bottom-half* code — called by interrupt handlers or soft interrupts. Top-half code, because it is called by a user process, has access to the the calling process' context, e.g., the stack. In contrast, bottom-half functions do not have access to any process context.

Thus, a bottom-half function must never call a (top-half) function that requires a process context, such as sleeping or locking operations. We would like to statically verify that this property holds when we create an operating system. Detecting this kind of error in a component-based system requires a global view, because whether a function is top-half or bottom-half depends on what functions it calls, and what functions call it. In other words, bottom-halfness is a *transitive* property: a function is a bottom-half function if it is called by a bottom-half function or by an interrupt handler.

Knit helps programmers deal with these kinds of "non-manifest" aspects of system composition with constraint annotations that describe the properties of unit imports and exports. Constraints can declared explicitly (e.g., that the functions within an export bundle are top-half) or by description (e.g., that the unit's exports have the same halfness as the unit's imports). At system configuration-time, Knit propagates constraints to ensure that all are globally satisfied. If a constraint is not satisfied, Knit prints a description of the error.

3 Units for Aspects

Units offer a clear solution to the problem of effectively expressing conventional modularity. *But we claim further that units are a proper foundation for developing the cross-cutting facets of a system in a modular fashion.* The essential ingredient provided by units is the ability to take an arbitrary component and "wrap" it with another component. In other words, because the links between components are explicit, we can modify a linking graph as we choose by interposing new units where they are needed to introduce aspect-like functionality. This interposition can effectively insert code at the beginning and/or end of every exported function from a set of components, or it can be used to uniformly apply a class extension — in the form of a mixin — to each class imported or exported by a set of components. In AOP terminology, the interfaces between units are the *join points* at which we may introduce code.

This approach to dealing with aspects is particularly compelling for systems and middleware code. As discussed in Section 1, sometimes no clear line can be drawn between modularity at the level of functional subsystems and modularity at the level of cross-cutting aspects. We therefore seek a framework that integrates these two facets of systems modularity. We expect the Knit approach to help deal with many different aspects of systems code, including concurrency, isolation, and real-time performance. We are currently exploring these aspects in the context of the OSKit [7], a large collection of components for building low-level systems.

Concurrency. In the OSKit, many components are single-threaded because they were extracted from single-threaded kernels such as Linux and BSD. However, as OSKit components, they are often used in multi-threaded environments. When combining many components with different concurrency requirements, it is extremely difficult to determine by inspection which components need to be wrapped, or even which combinations of components can be wrapped by a single concurrency wrapper. We expect Knit to help us solve these problems. By annotating interfaces and components, and then inserting concurrency adapters as needed, Knit will help us to build systems that are assurably correct with respect to their units' concurrency requirements.

Isolation. Flexible control of component isolation and protection is a novel property in the embedded and operating systems worlds. Even modern production operating systems implement at most three configurations of this important aspect: no protection, multiple threads sharing the same user process and address space, and entirely separate user processes. There are many reasons to want more flexible control: one may want to impose resource limits on particular components, isolate buggy components, or impose security restrictions on untrusted code. Knit will help us impose these kinds of isolation barriers on components in a flexible and reliable way.

Real-Time Behavior. Finally, we are interested in the specification and assurance of real-time behavior. One way to achieve real-time performance is to insert a small real-time kernel "under" an existing operating system: this is the approach taken by RT-Linux [2]. The real-time kernel interposes on the interrupt handling and manipulation routines in the host operating system, so that the real-time kernel receives all hardware interrupts. This approach is a natural fit with Knit's ability to interpose on component interfaces. A second approach to providing real-time behavior is to incorporate real-time functionality into the main operating system. In an embedded system, with a fixed set of real-time tasks, one would like to reason about the requirements of the tasks in order to see if they can be met: i.e., if the system has enough resources to schedule the tasks correctly. We intend to extend Knit's constraint language in the future to support this kind of reasoning, e.g., to deal with numeric constraints.

4 Improving Knit for Aspects

Units provide a plausible foundation for aspects-as-components, but we need to extend Knit in certain ways to handle aspects well. First, the linking language must be extended so that it is easier to describe new linking graphs (compound units) as modifications of existing units. Second, Knit must be enhanced to generate (or help generate) the code for "wrapper" and "adapter" units, to relieve programmers of this tedious task.

4.1 Unit Composition

Although explicitly specified linking for units has the advantage of giving the programmer complete control, it has the drawback of substantially increasing a programmer's work for linking a program. In the common case that a collection of components can be linked in only one way (because every import can match exactly one export among the set of linked units), the linking programmer's work is redundant. In the future, we will extend Knit to remove much of the burden of linking from the programmer, but still allow the programmer to take control when necessary. Finding the "right point" between explicit and implicit control will be a major issue addressed by this work.

The hierarchical composition of units is also problematic in certain situations. Hierarchical linking can provide valuable structure to a system, but it can also introduce repetitive imports and exports that are propagated across the levels of hierarchy. A related problem is that replacing a unit deep in the hierarchy is difficult. To replace a unit *U*, one might copy the definition of the compound unit *C* in which *U* appears, and modify the new compound unit *C'* to replace *U* with some other unit. Unfortunately, one must now "walk up the hierarchy" and do the same for *C* itself: i.e., copy the units in which *C* appears to substitute *C'*, and so on.

To make replacing components easier, a future version of Knit will support composition specifications that are defined in terms of existing composition specifications, but with certain components "overridden" by replacement components [1, 3]. A subtyping relationship on unit interfaces would ensure that such compositions can be statically checked by comparing the overridden unit's interface with the overriding unit's interface. It may even be possible to extend the subtyping relationship to the behavior aspects of a unit's specification [9].

4.2 Wrapper Units

In addition to new support for modifying unit compositions, we are now working to improve Knit so that it can generate certain kinds of wrapper or adapter units automatically. A wrapper unit is commonly used to implement certain idiomatic behaviors, e.g., to monitor, modify, or extend the wrapped unit's interface. While the basic unit composition mechanism allows a programmer to apply a wrapper to a set of components, manually wrapping each component is often too unwieldy in practice. Because many wrappers follow a pattern, the implementation of these wrappers can be automated. For example, many conversion tasks involve surrounding every function exported by a set of units with a common wrapper. (A single unit suffices for this purpose, instantiated once for each exported function.) Knit will provide a straightforward way to automatically instantiate the wrapper for every export. For more complicated cases, in which every wrapper must tailored to a specific use, we intend to incorporate a flexible code generation scheme into Knit, so that Knit can create the implementations and unit definitions for the wrappers. Our previous experience with a flexible stub compiler [4] will be useful in this area.

5 Conclusion

Knit is a new component definition and linking language, designed to handle the "manifest" and "non-manifest" aspects of systems and middleware code. Component composition in Knit acts like aspect weaving, and future improvements to Knit will increase its utility for aspect-oriented programming.

Availability

Source and documentation for our Knit prototype is available under `http://www.cs.utah.edu/flux/`.

References

[1] D. Ancona and E. Zucca. An algebraic approach to mixins and modularity. In M. Hanus and M. Rodríguez-Artalejo, editors, *Proc. Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 179–193. Springer-Verlag, 1996.

[2] M. Barabanov and V. Yodaiken. Real-time Linux. *Linux Journal*, 34, Feb. 1997.

[3] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, Mar. 1992.

[4] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proc. ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation (PLDI)*, pages 44–56, Las Vegas, NV, June 1997.

[5] M. Flatt. *Programming Languages for Component Software*. PhD thesis, Rice University, June 1999.

[6] M. Flatt and M. Felleisen. Units: Cool units for HOT languages. In *Proc. ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation (PLDI)*, pages 236–248, June 1998.

[7] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for OS and language research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.

[8] S. P. Harbison. *Modula-3*. Prentice Hall, 1991.

[9] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[10] J. G. Mitchell, W. Mayberry, and R. Sweet. *Mesa Language Manual*, 1979.

[11] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, Oct. 2000. USENIX Association.