

# Systems Need Languages Need Systems!

Robert Grimm  
New York University  
rgimm@cs.nyu.edu

## Abstract

Domain-specific language and compiler extensions can significantly reduce the complexity of systems, especially when written in C or C++. However, instead of the current variety of ad-hoc implementation strategies, which include preprocessor macros, C++ templates, and custom-built language processors, system builders need a uniform and general facility for realizing their own extensions. In this paper, we argue that macros can provide this solution, as they provide a concise specification of how to transform a program at compile time. We develop the requirements for a suitable macro facility, outline the structure of the corresponding macro processor, called `xTC` for eXTensible C, and explore the implications of starting to build it. We conclude that only a tighter integration between system and language efforts can help us achieve the benefits of language and compiler extensibility and cope with the complexity of modern systems.

## 1 Systems Need...

The utilization of programming language and compiler technologies in operating and distributed systems certainly is not new [2]; notable examples include the Pilot [13], Oberon [16], and SPIN [3] operating systems as well as the Fox Project's network protocol stack [4]. Yet the vast majority of system builders continues to rely on C (or C++) instead of fully type-safe or functional languages. When they need additional expressive power, they tend to add their own, domain-specific idioms to the base language by using a variety of ad-hoc implementation strategies, including preprocessor macros, C++ templates, or even custom-built language processors. However, the unfortunate result is that existing language and compiler extensions are mutually incompatible and that new extensions are hard to realize.

To illustrate the power of domain-specific language and compiler extensions and the range of implementation strategies, consider `libasync` [10] for building event-driven distributed services, `Capriccio` [15] for building multi-threaded servers, and `MACEDON` [14] for building overlay networks. Common to these projects is that they all build on C or C++, that runtime support is *not* sufficient, and that they *require* language or compiler support. At the same time, they differ significantly in

how they exploit language and compiler technologies. `Libasync` treats C++ templates (supplemented by several Perl scripts) as a macro system for providing a convenient interface to asynchronous programming. Its main benefit is that developers can write straight-line code instead of breaking functionality into separate event handlers, which obscure a program's control flow. In contrast, `Capriccio` builds on CIL [11] to perform domain-specific optimizations that allow the thread package to only allocate as much stack space as needed, thus avoiding the conservative allocation of large memory regions and providing better scalability than other thread packages. Finally, `MACEDON` provides a domain-specific language and preprocessor that extends C++ with support for specifying overlay protocols as state machines. By using this language, system builders can more easily develop specifications of their protocols, which are not only concise but also executable.

We believe that macros can provide a *general* and *uniform* foundation for enabling other system builders to readily achieve similar benefits. Macros are attractive because they provide a concise specification for how to transform a program at compile time. However, in departure from the C preprocessor and C++ templates, a suitable macro facility needs to meet four requirements. First, the macro facility has to be more expressive than traditional macro systems and support the expression of new *definitional* constructs, so that developers can directly express core concepts in their code. Examples include closures for event-based programming, which are illustrated in Figure 1 and generalize the function currying functionality provided by `libasync`, or state machines as provided by `MACEDON`. At the same time, examples are not limited to domain-specific constructs and also include generally applicable abstractions, such as modules, objects, or generics.

Second, the macro facility needs to be safe, so that the compiler can detect as many program errors as possible. Basic macro safety requires that macros are hygienic—that is, preserve the meaning of variable references in the macro definition as well as at the macro invocation site. Additionally, the macro facility needs to allow for the expression of new typing constraints, including the expression of new kinds of types associated with definitional constructs. Associating type information, including type rules, with language extensions is necessary be-

---

```

char* host = "www.example.com";
char* path = "/index.html";
dnslookup(host, void closure(ip_t addr) {
    tcpconnect(addr, 80, void closure(int fd) {
        ... write(fd, path, strlen(path)); ...
    });
});

```

---

Figure 1: Example use of a closure macro. The macro adds first-class procedures to C, letting developers write straight-line code in the presence of asynchronous callbacks. The example shows two such callbacks, to be invoked after completion of the DNS lookup and after establishment of the TCP connection to the resolved host.

cause it allows the compiler to guarantee the safety of the unexpanded code and to provide clear feedback in the case of errors, for example, when the closure macro has been applied on the wrong (types of) arguments. This stands in marked contrast to many existing macro systems, such as C++ templates, which perform type checking only after macro expansion. As a result, they provide little intuitive feedback about the source of errors while also exposing a macro’s desugared representation.

Third, the macro facility needs to create efficient code. In particular, the macro facility must support the expression of specialized expansions. For example, when expanding a `foreach` macro applied to an array, the resulting code should directly access the array members and thus be more efficient than the same macro applied on a generic list. Furthermore, the macro system must support code transformations that do not depend on new syntax but rather perform domain-specific optimizations, such as when analyzing the depth of call stacks as required by Capriccio. At the same time, we do not expect such a macro facility to support low-level, hardware-dependent optimizations. Furthermore, we believe that efficiency is easier to achieve by extending C than by replacing C with a more expressive and safer language such as Cyclone [9], since the implementation of language extensions can be very close to the implementation of comparable features in plain C.

Fourth, macros need to be expressible and composable at a fine granularity. To encourage the reuse of language extensions and the eventual creation of a default macro library akin to C++’s standard template library, individual macros should be as self-contained as possible and only add a single language feature, such as exceptions, classes, or closures. The macro facility can help macro writers achieve this by exposing a well-defined composition model, by making it easy to track dependencies between individual macros (such as a closure macro depending on a class macro), and by statically detecting conflicts between composed macros.

---

```

source {
    class $(id:Identifier) {
        $(members:MemberDeclarations)
    }
} target {
    typedef struct $(gensym(id,"object")) {
        void *vtable;
        $(select("/FieldDeclaration", members))
    } * $id;
    ...
} newtype {
    kind = "Object",
    class = id,
    fields = createmap(
        select("/FieldDeclaration/Identifier",
            members),
        types(select("/FieldDeclaration/Type",
            members))),
    methods = ...
}

```

---

Figure 2: An example class declaration macro. The macro specifies the source syntax (grammar rules are omitted for brevity), the corresponding transformation, and the type rule. The transformation expands the declaration into a C structure containing the fields (shown) and another C structure for the vtable (not shown). The type rule constructs a record representing the structural information of the class.

Despite a considerable body of previous work (see [5] for a comprehensive comparison between several representative macro systems), no existing macro facility meets all four requirements. Notably, we are aware of very little work on extensible type systems and on composing language extensions.

## 2 Languages Need...

To meet the four requirements, we are exploring a new macro system for C, called `xtc` for `eXTensible C`. `xtc` macros consists of grammar, abstract syntax tree (AST) transformation, and type rules. We expose the *entire* grammar for modification to maximize expressivity. Grammar rules support the addition and removal of syntactic constructs at the granularity of a single alternative in the targeted production’s choice. They may also introduce entirely new productions. We represent the actual macro expansions as AST transformations because a program’s AST provides a concise representation of its structure. As for other macro systems, the corresponding rules are expressed as templates that mix literal syntax with the macro language’s pattern expressions (which, in turn, may included arbitrary nonterminals). We support type rules to provide domain-specific safety guarantees. Type rules can be used, for example, for declaring new types, for expressing subtyping relationships, and for deducing the overall type of an expression contain-

---

```

source {
  $(exp:Expression!Object) . $(id:Identifier)
} target {
  $exp -> $id
} type {
  select("/fields/$id", typeof(exp))
}

```

---

Figure 3: An example field access macro. The macro specifies the source syntax, the corresponding transformation, which expands the field access into a C structure access, and the type rule, which looks up the field’s type in the class’s type record. The kind constraint `!Object` ensures that the rule is applied only to field accesses for objects but not C structures or unions.

ing a newly introduced construct. Two example macros illustrating `xtc`’s *preliminary* syntax for AST transformation and type rules are shown in Fig. 2 and 3.

Like many other macro systems building on AST transformations, `xtc`’s templates mix literal syntax with pattern expressions. However, to support new definitional constructs, `xtc`’s templates require additional expressive power. For example, a class declaration includes zero or more member declarations inside the class body, with each member declaration representing either a field or method declaration. When specifying the corresponding expansion, the macro writer needs to select the field declarations separately from the method declarations and splice them into a C structure representing the object (and, similarly, splice the method declarations into a C structure representing the vtable). To address this need for querying a program’s AST, `xtc` includes a tree query language. As illustrated in Figure 2, this language builds on XPath, which is a well-established standard for querying XML, models each document as a tree of nodes, and combines the simplicity of path-like selectors with the flexibility of additional filter expressions. However, instead of a generic document processing library, we include a domain-specific library for analyzing ASTs.

In making the type system extensible, `xtc` needs to provide macro writers with some way of representing the structural information about new kinds of types. For some kinds, this structural information is trivial. For example, rational numbers, from the type system’s view, are all the same and thus introduce only a new type name. Java-like objects, on the other hand, require information about the class name, the superclass, implemented interfaces, and the declared fields and methods. To capture this information, `xtc` type rules can use collections of name/value pairs, which may be nested within each other, as well as lists of values. The advantage of this representation is that it can readily be repre-

sented as a tree. As a result and as shown in Figure 3, type rules can also use our tree query language.

`xtc`’s macro selection builds on Maya’s multiple dispatch model [1] and supports dispatch on nonterminals, types, and kinds of types. As discussed in Section 1, dispatch on types is necessary to support specialized expansions that are more efficient than the more general base case. As illustrated in Figure 3, dispatch on kinds is necessary to support new definitional constructs that have the same syntax as other constructs in plain C.

Finally, we expect that typical `xtc` macros are small and provide well-defined building blocks for extending C. As a result, developers will use several macros at the same time, composing them to provide the desired extended language. However, fine-grained macro composition also requires support by the macro system. To this end, `xtc` supports the grouping of macros into larger language extensions and the explicit declaration of dependencies between extensions, comparable to a lightweight module system in many programming languages.

While we believe that the macro facility just described will make the benefits of language and compiler extensibility readily available to system builders, it also requires a substantial implementation and engineering effort. We can deflect some complexity by implementing `xtc` as a source-to-source transformation system, leaving traditional optimizations and code generation to an existing C compiler such as `gcc`. Comparable to CIL [11] and Polyglot [12], we have also created our own toolkit for building extensible source-to-source transformers. The toolkit includes support for representing and traversing ASTs as well as an easily extensible parser generator, which leverages parsing expression grammars [7] to provide more flexibility than LR and LL parser generators [8]. Our toolkit already parses and pretty prints the entire Linux 2.6.10 kernel. Nonetheless, we are faced with a considerable system building effort to support a macro facility that builds on source and target templates, includes explicit tree queries, requires access to semantic information such as types, and selects macros through multiple dispatch.

### 3 Systems!

To bridge the gap between the developer-visible macro language and our basic source-to-source transformer toolkit and, consequently, to reduce the complexity of `xtc`, we are currently extending our toolkit with a facility for querying and transforming abstract syntax trees. The basic idea is to let tool developers, including us for `xtc`, declaratively specify how to manipulate a program’s AST instead of explicitly programming the corresponding AST analysis and transformation steps—much like embedded databases simplify the manage-

ment of persistent state.

Our AST query facility builds on the same tree query language already exposed to macro writers, with additional support for binding variables, accessing semantic information including types, and replacing AST fragments by instantiating simple AST templates. With the query facility in place, `xtc` can convert source and target templates into queries, which may incorporate explicit `select` statements, and into AST templates, respectively. Type rules can be converted into expressions that update or verify the associated semantic information, depending on whether the rule is a `newtype` or `type` rule.

To provide access to a program's semantic information, the query facility will leverage our toolkit's support for per-node metadata and annotate AST nodes with their type and kind information. Furthermore, to push this information throughout a program's AST, our query facility will include an extensible type inference engine. Since C-like languages rely on explicitly declared types (unlike, say, Standard ML or Haskell), inference itself is relatively straight-forward. The challenge is to support the introduction of new kinds of types. We plan to address this challenge by using a common interface for all types, which specifies, for example, the definition of type equality and subtyping relationships.

By building on our query facility, `xtc`'s multiple dispatch model can directly be implemented as a set of related queries and thus does not need any additional implementation mechanism. All queries are scheduled together, which improves performance when compared to performing queries individually, as common subexpressions only need to be evaluated once. It also enables the query facility to only trigger the transformation for the most specific match, with the result that a more specialized macro, such as `foreach` operating on arrays, will take precedence over the more general version.

The above discussion illustrates why a facility for querying and transforming ASTs is crucial for realizing `xtc`. First, to support new definitional constructs, the macro processor already requires some support for querying abstract syntax trees. Second, to implement the multiple dispatch model for selecting macros, the macro processor, again, requires a facility for querying abstract syntax trees. By building on a sufficiently expressive query facility, a single mechanism can meet both needs and thus reduce complexity. The discussion also illustrates why a generic tree querying and transformation facility, such as an XSLT processor for transforming XML documents, is not sufficient: To be suitable for `xtc` and other language processors, the query facility needs to incorporate semantic information that is not directly contained in the tree, notably by performing type inference.

To validate our toolkit and query facility as a sub-

strate for easily extending C, we are not only using it for implementing `xtc` but also as the basis for an aspect-enhanced C compiler [6]. More specifically, we are exploring how to improve the maintainability of the Linux kernel. The problem is that current tools for distributing and applying kernel variants, notably `diff` and `patch`, operate only at a textual, line-by-line level. As a result, large-scale patches are hard to maintain and apply, as the kernel changes considerably even across minor releases. Our work seeks to create a more suitable alternative, which represents kernel variants as aspects, thus elevating them from a textual to a syntactically meaningful representation, and provides semantic analyses to automatically detect conflicts between different patch sets, thus simplifying the creation of kernels that combine several variants.

In summary, macros have the potential to significantly reduce the complexity of modern systems by letting developers readily realize domain-specific language and compiler extensions. But a suitable macro processor, which meets the requirements of expressivity, safety, efficiency, and composability, is a complex system in and of itself. Consequently, we need to approach the implementation of the macro processor just like other large-scale system efforts and focus on making the infrastructure extensible as well as on providing intermediate services, such as an AST query facility, that reduce the implementation effort for the macro processor and other programming tools alike. To put it differently, only an increased integration of systems and language practice can help us cope with the complexity of modern systems. ❄

## Acknowledgements

We thank Marc Fiuczynski, Benjamin Goldberg, Joe Pamer, and the anonymous reviewers for their discussions and feedback. This work is funded in part under NSF grant CNS-0448349. More information on `xtc`, including a source release, is available at <http://www.cs.nyu.edu/rgrimm/xtc/>.

## References

- [1] J. Baker and W. C. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation*, pp. 270–281, Berlin, Germany, June 2002.
- [2] H. Bal, J. Steiner, and A. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, Sept. 1989.
- [3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 267–284, Copper Mountain, Colorado, Dec. 1995.

- [4] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in Standard ML. *Higher-Order and Symbolic Computation*, 14(4):309–356, Dec. 2001.
- [5] C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 31–40, Portland, Oregon, Jan. 2002.
- [6] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. patch (1) considered harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, New Mexico, June 2005.
- [7] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pp. 111–122, Venice, Italy, Jan. 2003.
- [8] R. Grimm. Practical packrat parsing. Tech. Report TR2004-854, New York University, Mar. 2004.
- [9] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pp. 275–288, Monterey, California, June 2002.
- [10] D. Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pp. 261–274, Boston, Massachusetts, June 2001.
- [11] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, vol. 2304 of *Lecture Notes in Computer Science*, pp. 213–228, Grenoble, France, Apr. 2002. Springer.
- [12] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, vol. 2622 of *Lecture Notes in Computer Science*, pp. 138–152, Warsaw, Poland, Apr. 2003. Springer.
- [13] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, Feb. 1980.
- [14] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the 1st ACM/USENIX Symposium on Networked Systems Design and Implementation*, San Francisco, California, Mar. 2004.
- [15] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 268–281, Bolton Landing, New York, Oct. 2003.
- [16] N. Wirth and J. Gutknecht. *Project Oberon—The Design of an Operating System and Compiler*. Addison-Wesley, 1992.