A declarative approach to extensible interface compilation

Nicholas FitzRoy-Dale
National ICT Australia and University of New South Wales
Sydney, Australia
nfd@cse.unsw.edu.au

Abstract— **In microkernel-based operating systems, source-to-source compilers generate code to ease the process of marshaling data for communication via message passing. However, the rule of thumb for these *interface compilers* seems to be "simple, extensible, efficient output – pick any one". I argue that the major cause of extensibility-limiting complexity in interface compilers comes from the source-to-source transformation code itself, and this complexity is primarily a result of the difficulties inherent in supporting multiple targets. I describe a specification-based approach for generating interface compilers, discuss the advantages of such an approach over a procedural approach, and outline a proposed implementation, with particular reference to the advantages of a specification-based approach to interface compilation in terms of flexibility and extensibility.**

I. Introduction

With any systems programming task come situations involving code that could easily be machine generated. A particularly good example of this in microkernel-based systems is in the code required for communication of messages across an interface boundary. Such code is not strictly boilerplate, because it varies with the data layout of the message being sent, but the interface-specific differences can easily be described programmatically. Using a code generator is not necessarily the right choice: adding another layer of abstraction to the communication process makes debugging more difficult and encourages new programmers to ignore the intricacies of the underlying interface (the so-called *leaky abstraction* problem[1]). However, code generators make up for these shortcomings in many ways. For example, several classes of bugs are not possible in generated code, and all code making use of the same interface will communicate across it in the same way.

Automating the process requires a tool capable of producing code from a specification, usually an *interface compiler.* This specialised tool parses an interface specification, usually in a form of Interface Definition Language (IDL), and produces output in the language used to implement the rest of the system (the *target language*), typically in the form of *stub* functions. Code on the caller side of the interface uses these stubs like any other function: the work of packing function parameters into an appropriate location in memory (*marshaling*), communicating using an appropriate operating system primitive, and extracting the results from memory and returning them to the calling function (*unmarshaling*) is performed by the stub. Interface compilers produce appropriate *mirror* stub functions for the service side of the interface, and may also produce skeletal code for a server loop implementing the interface.

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

Interface compilers are used in many areas of computing. Servers and distributed systems may make use of a *component system* of which stub compilation is only a very small part, such as an implementation of CORBA[2], COM[3], or JavaBeans[4]. All these component systems support distribution across a network in addition to other "enterprise" features such as load balancing and quality of service. In microkernel-based systems, by contrast, space and efficiency constraints tend to result in the need for safe cross-domain communication without the overheads of a traditional component system. In this situation, most of the work is performed in stubs produced by an interface compiler.

Interface compilers for microkernel-based systems are the focus of this paper. In addition to missing many of the features described above, typical interface compilers in this category differ from their enterprise equivalents in two major ways: they generally produce simpler code, because the microkernel performs the tasks of message-passing and queueing and thus acts as a (partial) object request broker; and they are more likely to be required to produce multiple types of output, even for the same microkernel, because the primary concern of microkernel-focused interface compilers is to produce fast, as opposed to feature-complete, code.

Historically, these interface compilers have been heavily tied to their attendant microkernel. For example, the Mach Interface Generator (MIG)[5] accepts a Mach-specific interface specification, produces highly Mach-specific code, and could not reasonably be used with another kernel. More recently, interface compilers capable of accepting various IDLs and supporting multiple target languages have emerged. These include the Flexible IDL Compiler Kit (Flick)[6], which supports IDLs such as CORBA IDL and ONC RPC at the frontend, and targets Mach[7] and various incarnations of L4[8] at the backend. Flick achieves this modularity through a series of programmer-visible intermediate languages (five in total) which can be operated on independently.

Changes in the design of an interface compiler are typically made either to improve performance or improve gener-

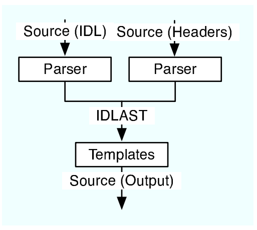

Fig. 1. Stages in the Magpie interface compiler

ality. However, interface compilers for microkernels remain remarkably difficult to adapt to changing interface requirements. There are three major reasons for the difficulty: firstly, they tend to remain tightly-coupled to a small selection of kernels; secondly, the core interface-generation routines tend to be difficult to modify; and, finally, representation of target code is primitive compared with that offered by source-to-binary compilers.

In the rest of this paper I describe extensibility issues and their various solutions in current interface compilers. I then propose an alternative interface compiler design, making use of specification-directed transformations, and discuss the advantages and disadvantages of this approach. Although I refer to several popular interface compilers, the focus is on adding a declarative layer to Magpie, an interface compiler used with NICTA L4.[9]

II. Interface compilers

Like source-to-binary compilers, interface compilers are all composed of at least a *frontend*, containing the parser for the specification language, and a *backend*, containing the code generator for the target system. It is common to make use of at least one intermediate representation. Figure 1 shows the major stages of the Magpie interface compiler.

Source-to-binary compilers may support multiple platforms, but the targets (platform-specific assembly languages) are all very similar; intermediate stages therefore output some variety of three-address code[10], which is by design quite homogeneous. The intermediate stage of an interface compiler, however, must target some higher-level language, such a C, working with a kernel or library API. Unlike processor assembly languages, kernel or library APIs are very dissimilar, even when designed for the same task (i.e., communication between threads). Designing an effective intermediate stage for an interface compiler is thus simply a harder task, because of higher degree of cross-platform variation. A common response in interface compiler design has been to blur the separation between intermediate stages and final code generation, resulting in intermediate stages which are complicated, inflexible, or both.

Further complicating the task of the interface compiler are the frequently-changing requirements of applications and the kernel, a particularly severe problem for research systems. For example, NICTA L4 has undergone several internal API changes, each one necessitating an equivalent change to the interface compiler. The difficulty of implementing these changes in the popular IDL4 interface compiler has led to the development of Magpie, an interface compiler based on a flexible template system. The partial success of Magpie in this regard is the prime motivation for the work described in this paper.

A. Code transformation

The real work of an interface compiler is performed by the intermediate stage known as the *generator*, which analyses the internal representation of the interface definition and produces output (either in the target language, or in an intermediate format). In modern interface compilers, the generator forms a third stage between the parser and code output stages, allowing for some degree of flexibility with regard to input and output formats. Nonetheless, making sizeable changes to the format of the output requires modifying the generator, and here choice of language and generator design become important: in many common interface compilers for microkernels, such as Flick, IDL4, and DICE, making even simple changes requires changing a nontrivial amount of code.

B. Code output

Code output is widely regarded to be an easy problem and is thus given little attention during interface compiler development. Indeed, it is easy (in the sense that little design is required) to create a simple interface generator that generates appropriate code for a fixed interface, tied to a single microkernel. However, there are several problems with this approach that both limit the scope of interface compilers and increase their maintenance overhead.

Typically, an interface compiler backend is composed of a series of `printf` statements which together produce the entire stub function or module. This approach is simple to implement, and the intent of the code is obvious, but it is difficult to extend and to test. Alternatively, the backend may assemble a syntax tree in the target language, which is then *walked* as a final step to produce source code. This approach at least provides some reassurance that the generated code will be syntactically correct, but the usual method of assembling the tree in a procedural language, calling class constructors, is difficult to follow, and difficult to extend.

Before considering parameterised templates it is important to consider boilerplate code, that is, code which does not change between invocations of the compiler. Generated code typically contains a lot of boilerplate in the form of header comments, error-handling code, helper routines, etc. A typical procedural approach to generating this sort of code is simply to insert it at appropriate points when producing output. This looks ugly for a system that uses `printf`-style output, and practically unreadable (if it occurs in large chunks) in a system that generates a concrete syntax tree on-the-fly. In either case, boilerplate generation is simply noise for a maintainer attempting to understand what the interface compiler does. The situation gets worse if the programmer wishes to extend the system,

```
addTo(result, new CASTDeclarationStatement(
  new CASTDeclaration(
    new CASTTypeSpecifier(
      new CASTIdentifier("L4_ThreadId_t")),
    new CASTDeclarator(
      new CASTIdentifier("_dummy"))))
);
```

Fig. 2. Variable declaration, IDL4

```
1 L4_MsgTag_t _result;
2 /*-run(templates.get(
    'client_function_body_pre_ipc_defs'))-*/
  ...
3 /*-run(templates.get(
    'client_function_body_pre_ipc'))-*/
4 _result = L4_Call(_service);
5 /*-run(templates.get(
    'client_function_body_post_ipc'))-*/
```

Fig. 3. An example of the Magpie templating language

perhaps with a new output mode. She is then presented with two options: to copy the code, creating the opportunity for unwanted divergence in the future; or to refactor the code to be more fine-grained, making flow less obvious and providing no guarantee that additional changes will not necessitate further refactoring in the future.

The difficulties associated with code generated piecemeal are magnified for sections of parameterised code, i.e., the sections of code in which the generator actually performs work. Using a syntax tree approach, a single line of (relatively-simple) generated code consumes approximately six lines of generated AST (abstract syntax tree) in IDL4. Figure 2, extracted from IDL4, creates a new type instance using multiple classes to construct an abstract syntax tree, which is later walked to produce output. The example in the figure is the equivalent of the C code `L4_ThreadId_t _dummy;`.

C. Magpie

The Magpie interface compiler was developed in recognition of these code-generation problems. Magpie uses a simple templating system, interspersing control code and the target language. The design goals were to keep the backend simple to maintain for developers who wished to make use of alternative interfacing techniques but did not wish to become intimately familiar with Magpie's code base, and to maintain code flow of the target language as much as possible, thus making the generator and backend easy to understand and, in turn, easy to extend. Magpie was not the first interface compiler to make use of templates. Flick supports its own templating system with the apparent design goal of reducing the amount of non-parameterised code present in the existing interface compiler. Because they are relatively new, Flick's templates are not supported by most backends.

A small example of the Magpie templating language is

```
int notifymask;

notifymask = set_notifymask(0);
L4_Call(...);
set_notifymask(notifymask);
```

Fig. 4. Code to manage async IPC in NICTA L4

shown in Figure 3. The templating command language is embedded inside comments within the target language (Python within C, in this case). The code shown executes the `L4_Call()` function, which performs synchronous IPC in L4. Before and after the call, additional templates are executed using the `run()` function (lines 2, 3, and 5). These templates may generate code that performs additional work to support the IPC operation. For example, in NICTA L4, asynchronous IPC notifications must be disabled on the client side prior to a synchronous IPC call. Code generated by the `client_function_body_*` functions thus saves the old state of asynchronous IPC notification, disables asynchronous IPC notification, and re-enables it after `L4_Call()` completes. The C code for this functionality is show in Figure 4.

The `run()` command is a convenient extensibility mechanism, but convolutes the flow of the template, contrary to the stated design goals (specifically, understandability of the backend). In the example given above, four separate source files are required to properly disable and re-enable asynchronous IPC.

Magpie's templating approach is at best only partially successful. Code flow is preserved (the above example notwithstanding), but the templated code is difficult to read, and does not get significantly easier as one becomes more familiar with the system. The templates are not stand-alone: to understand the system one must also be familiar with the procedural-code generator with which the templates communicate and, in some cases, the abstract syntax tree containing the parsed data. Perhaps the best proof of both the success and failure of Magpie is the fact that after more than a year of use, many different templates were successfully developed to accommodate changing requirements, but the sole developer of new templates was the implementor of Magpie.

III. A specification-based approach

The interface compilers discussed in the previous sections all use a procedural approach to perform code generation. That is, a program written in a procedural language (commonly C or C++) examines an AST created by the frontend and generates code in the target language, if the compiler does not include a separate generator, or an intermediate representation, if it does. This approach manages to be both too general-purpose and too inflexible: a programmer wishing to modify the behaviour of the stub code must become very familiar with interface-compiler internals, and must modify a nontrivial amount of code to extend the system.

The use of a procedural language to write the genera-

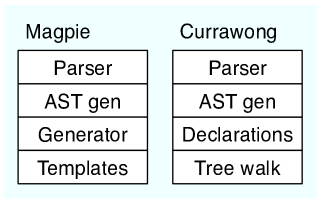

Fig. 5. Magpie and Currawong

```
typedef unsigned int counter_t;

type counter_t  (1, example.h:1)
  (meta_type) = ['alias']
  (indirection) = ['']
  target  (2, ?:?)
    type unsigned int (backref)
```

Fig. 6. A simple C typedef and its AST representation

tor section of the interface compiler is an obvious choice, because the rest of the interface compiler is typically implemented in a procedural language. However, a procedural approach results in complicated, verbose generator code. The core of an interface compiler is essentially a tree-to-tree transformation. In this section, I demonstrate that using a declarative language confers several benefits over the standard approach, particularly in the areas of code size, understandability, and extensibility. The design comprises a set of modifications and extensions to the Magpie interface compiler, resulting in a new interface compiler named Currawong.

A. Currawong and Magpie

Currawong replaces most of Magpie's generator and templating sections, as shown in Figure 5. Magpie's frontend parser and generators are preserved. The intermediate and backend layers are combined to form a declarative processing layer. The output from this layer is essentially an AST, so a final small stage walks the tree to produce output.

B. Procedures and declarations

As discussed above, interface compilers are complicated at least partly because of the strong relationship between the target language and compilers for that language – such as C and the `gcc` compiler. In short, the specification of any language mapping must necessarily conflate interface-level details, such as the layout of data in memory, and language-level details, such as the appropriate casts, bitwise shifting, and logical operations required to create this layout. This problem cannot be avoided completely – at some point, we must speak the target language – but its impact on generator extensibility can be minimised by separating the levels of specification. Referring to the previous example: if a given platform requires that parameters be marshaled into a certain location in memory, it is plausible that one may desire to change the marshal location of a given 64-bit integer in an interface-defined function, but less likely that one would wish to change the fact that, in C, the low-order 32 bits of this word may be accessed using "`theword & 0xFFFFFFFF`".

A common response to the specification requirement has been to perform exactly the separation described above. The Flick interface compiler, for example, uses five separate abstract representations for language-level and interface-level requirements. However, these five abstract representations have been determined by the implementors of Flick. They can, therefore, not be easily modified to accommodate level-spanning problems, and are thus inflexible: to again refer to the parameter-passing example above, one may make arbitrary changes at the level of determining the names and types of parameters to be marshaled, but the layout and organisation of parameters is only accessible at the lowest level, `CAST`, which essentially encodes a C++ abstract syntax tree. Other interface compilers tackle the same problem using a highly-stratified generator and rely on the language features of their interface compiler's language to supply extension. For example, the IDL4 and DICE compilers, written in C++, stratify their generators into the generator proper and a smaller section that separates messages into *channels*, allowing message order to be changed through interface inheritance.

In the Currawong system, the generator applies a series of transformations to a *code template* written in the target language in order to transform the template into an appropriate stub function. The "meat" of the transformation system is specified declaratively in Prolog and, due to the template, only the transformations necessary to customise the template to a particular interface are specified. These two characteristics combine to make the customisable aspects of the transformation system very compact and understandable.

C. Type management

In order to compile an interface definition, it must first be parsed and converted to an easy-to-manipulate form. In the Magpie interface compiler, the primary data structure is an abstract syntax tree produced by the frontend. An example of this structure is shown in Figure 6, with the C code that it represents. Magpie can make use of any type declaration presented in its frontend and, therefore, types defined in the interface definition file, and types defined in any included C files, are interchangeable.

Representing Magpie's AST in Currawong is a simple matter of transforming the AST, producing idiomatic Prolog nested structures. An example is shown in Figure 7. Some simplification may take place: AST nodes representing alias types (such as those defined by `typedef` in C) include a direct reference to the target type of the alias (strictly speaking, the "AST" is actually a directed acyclic graph) but the converted representation does not. Node *attributes* (`meta_type` and `indirection` in this case) are discarded or converted, and line number information is not preserved. Nonetheless, this form of specification is very powerful, because it allows one to use Prolog's unification facility in the expected way to perform pattern-matching.

```
type(alias(counter_t, name(unsigned int)))
```

Fig. 7. A simple C typedef, Prolog syntax

For example, all C `typedef` types may be matched using the Prolog term `type(alias(Name, Type))`.

D. Specification language

The Currawong specification language is a declarative representation of the generator present in other interface compilers, such as Flick, IDL4, and Magpie. A generator in the Currawong syntax is a series of declarations, each one representing a single code transformation.

In general, a code transformation may be described as consisting of three parts: a *match rule* that locates the desired portion of code, a set of *requirements* which serves to determine whether the code is correct with regards to the transformation, and a *transformation rule* which may be followed to transform the matched code and make it satisfy the requirements. In Currawong, these three parts are separated using a minor extension to Prolog syntax, as per the example shown in Figure 8: a syntax marker consisting of two colons separates the match rule from the requirements. In other regards, the syntax is Prolog. In the current experimental prototype, Currawong includes its own interpreter for this language, which is essentially a custom Prolog implementation with the above extension. A future, more complete, version of Currawong will support one or more open-source Prolog implementations, such as SWI-Prolog or GNU Prolog, instead.

Matching of rules and requirements is performed using the Prolog unification mechanism. A transformation is considered complete when, for each item matched by a match rule, the corresponding set of requirements also matches. Referring to the example in Figure 8, any set of parameters is matched, and the parameter list is bound to the Prolog variable `P`. Then `list_head` is called, which ensures that the first element of `P` is the parameter `L4_Word_t _service`.

If a match rule is encountered for which the corresponding requirements do not match, *code transformation* is performed. Code transformation proceeds as follows:

1. Take the first requirement from the list of requirements.
2. Find the corresponding transformation rule, which is the rule whose functor starts with `match_` and ends with the functor of the requirement under consideration. The corresponding transformation rule must have an arity one greater than that of the requirement under consideration – the final parameter is taken to be the transformed rule, and all other parameters correspond.
3. Unify any free variables in the transformation rule. Replace the matched portion with the result generated by the transformation rule.
4. Remove this requirement from the list.
5. Repeat this procedure until there are no more requirements.

Referring again to 8, if the paramter list does not contain `L4_Word_t _service` as a first parameter, the list is mutated by a call to `make_list_head`, and the AST is updated.

```
parameters(P) :: list_head(P,
    parameter('L4_Word_t', '_service').

list_head([H|T], H).
make_list_head(List, H, Result)
    :- Result = [H|List].
```

Fig. 8. An example of Currawong rule language

Note that the match rule and requirements reside in the same clause, on either side of the double-colon separator – collectively named the match/requirements clause. However, the transformation rule is a separate clause. This is because it is appropriate that the transformation rule, being far more general-purpose than the match/requirements clause, reside in a separate system-wide library of manipulation rules.

Given that Currawong parses a template and applies its transformations to the template, there is a great potential for reduction of code size via elimination of boilerplate code. In effect, Currawong statements act as a sort of aspect language for aspect-oriented programming (AOP)[11], specifying transformations through matching. Prolog is an ideal choice for an aspect language, because it is well-suited to the types of tasks, specifically searching, that form a large part of any aspect language.

IV. Examples

This section describes some examples for which a declarative code-transformation approach is well-suited.

A. L4 notification mask

Many superficial changes to non-parameterised code can be made simply by changing the code itself. For example, asynchronous notification was recently added to NICTA L4. It is undesirable for asynchronous notifications to arrive during a synchronous IPC, so they should be disabled prior to the IPC and re-enabled subsequently, as discussed in Section II-C. An example Currawong implementation is shown in Figure 9. Although this declaration effectively generates code to manage asynchronous IPC, it is essentially a correctness check, and can be used as such.

It is worthwhile to compare the declarative specification with the implementation of the same functionality in Magpie (Figure 3). In Magpie the code is distributed over several templates, including “hooks” (the `run()` command) in the base template, but the Currawong implementation has the form of an aspect in the AOP sense: both the base code and the aspect are well-separated, and significant changes may be made to the base code without any knowledge of the asynchronous support. In fact, the sorts of changes that would require knowledge of the asynchronous IPC support are those that would require rewriting the support anyway, such as a decision not to use `L4_Call()` in the base code.

```
context(expression(call(L4_Call, _)),
    Before, After) ::

    list_contains(Before, expression(
        equals(Var, call(set_notifymask(0))))),
    list_contains(After, expression(
        call(set_notifymask(Var)))).
```

Fig. 9. A currawong declaration: managing asynchronous IPC

```
 1 function(iguana_pd_pypd, _, Body) ::
 2     % Ensure that we declare the static.
 3     list_contains(Body, expression(typeinst
 4         ('static uintptr_t', 'mypd', 0))),
 5
 6     % Return early if the static is
 7     % initialised.
 8     context(expression(call(L4_Call)),
 9         Before, After),
10     list_contains(Before, if(expression(
11         notequals(mypd, 0),
12         expression(return(mypd)))),
13     % Initialise the static after the call.
14     list_contains(After, expression(
15         equals(mypd, __retval))
```

Fig. 10. An interface-specific optimisation rule

B. *Interface-specific optimisations*

The previous examples demonstrated flexible creation of generic interface compilers. However, not all interfaces are the same, and some may benefit from interface-specific customisations – optimisations in particular.

A simple example occurs when caching. The Iguana L4 OS personality separates threads into *protection domains*, where two threads in disjoint protection domains may not share data. A thread's protection domain identifier is represented by a nonzero integer which threads may obtain using the `iguana_pd_mypd()` function. This integer does not not change over the thread's lifetime. It might make sense, then, to include a declaration to perform caching on the `iguana_pd_mypd` interface function, which returns a thread's protection domain identifier. Such an optimisation rule is shown in Figure 10. This rule declares a static variable to cache the result of an IPC call (lines 3 and 4), checks to see if it can return the static before making the call (lines 8 to 12), and sets the static to the result of the call afterwards (lines 14 and 15). The Prolog representation of the C code is somewhat verbose – a possible solution to this aesthetic problem is discussed in Section V.

V. Related work

The description of Currawong above includes an example of interface-specific optimisation in C. Approaches for non-microkernel-based systems have focused less on per-interface optimisation and more on per-language optimisation. The Concert Signature Representation[12] does not make use of an explicit interface definition language, but allows definition of an interface using the target language's own definition sub-language (for example, function prototypes in C) with extensions. It then relies on a declarative mini-language to specify the type and location of each parameter, in a similar fashion to Currawong's declarative language. The experimental interface generator Mockingbird[13] arrives at a similar solution in a different way. Recognising that the ideal presentation of an interface in C++ is very different to an ideal presentation of the same interface in Java, Mockingbird allows programmers to hand-define the ideal interface in both languages, and ensures correctness by compiling both interfaces to its own unambiguous internal interface definition language and ensuring that the internal representations for both interfaces are the same.

Microkernel-focused systems concentrate more on per-interface customisation. Ford et al. [14] discuss the importance of separating interface *presentation* (the programmer-facing side of the interface; essentially the stub code signature) from the interface *contract* (the data that must be transferred) and describe a small specification language to customise the presentation for an interface generator for the Mach microkernel.

Currawong is not the first design to make use of declarations to perform program modification, known as *logic metaprogramming.* The TyRuBa system[15] is a tool for aspect-oriented programming in Java (a *weaver*) in which aspects are specified in a variant of Prolog. The authors claim that this approach allows easy extensions of the aspect language. An interesting feature of TyRuBa not currently incorporated into the Currawong design is the ability to embed portions of the target language directly into aspect declarations. Judicious use of such a feature could simplify blocks of code with few external dependencies, such as the optimisation example given above. If the target code were parsed by a Currawong implementation, it could also simplify examples such as the first one, allowing all rules to be written in plain C.

VI. Limitations and future work

As mentioned above, information is lost when type information is converted from the AST representation supplied by the parser to Prolog-style declarations. This information, which includes line number information, may be useful when reconstructing modified files. This information may be restored in several ways. Perhaps the most obvious is simply to store the information as additional attributes in the data structure representing a Prolog atom or node. The extra information could then be accessed using built-in predicates.

Haeberlen et al.[16] implemented a number of optimisations when designing the L4-specific interface compiler IDL4. While some of these (such as the direct-stack transfer) are architecture-specific, it would be informative to implement these optimisations in the framework of Currawong, to determine which are feasible in that context. Although I suspect the completed Currawong implementation to run relatively slowly in its first incarnation, the runtime speed of the generated stub code should be entirely dependent on the input passed to Currawong – transformations performed inside Currawong should not have an impact on stub performance.

Although most of the sections necessary to implement the Currawong extensions to Magpie are complete, some implementation work is still required to produce a working, testable interface compiler.

VII. Conclusion

Interface compilers, particularly interface compilers for microkernel-based systems, have some unusual requirements for which traditional compiler techniques are not particularly well-suited. The requirement to support a large array of targets, and to do so using a relatively high-level language, creates unique problems and has produced a variety of novel solutions. The use of a declarative language, combined with aspect-oriented techniques, has the potential to reduce the complexity and overall size of an interface compiler generator, and is worthy of further exploration and implementation.

References

[1] Joel Spolsky, "The law of leaky abstractions," `http://www.joelonsoftware.com/printerFriendly/-articles/LeakyAbstractions.html`, 2002.

[2] Object Management Group, "CORBA 3.0.3, Common Object Request Broker Architecture (Core Specification), 2004-03-01," 2004.

[3] Microsoft Corporation and Digital Equipment Corporation, *The Component Object Model Specification*, 1995.

[4] Sun Microsystems, "Java beans: A component architecture for Java," 1996.

[5] Open Software Foundation and Carnegie Mellon University, *Mach 3 Server Writer's Guide*, Jul 1992.

[6] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom, "Flick: A flexible, optimizing IDL compiler," in *SIGPLAN Conference on Programming Language Design and Implementation*, 1997, pp. 44–56.

[7] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "MACH: A new kernel foundation for UNIX development," Tech. Rep., Carnegie Mellon University, Computer Science Dept., Pittsburgh, PA, USA, 1986.

[8] Jochen Liedtke, "On micro-kernel construction," in *SOSP 1995*, Copper Mountain, CO, USA, Dec. 1995, pp. 237–250.

[9] Gernot Heiser, "Secure embedded systems need microkernels," in *;login:, USENIX*, Dec. 1995.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[11] Gregor Kiczales, John Lamping, Anurang Mendhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-oriented programming," *ACM Comput. Surv.*, vol. 28, no. 4es, pp. 154, 1996.

[12] Joshua S. Auerbach and James R. Russell, "The Concert Signature Representation: IDL as intermediate language," *ACM SIGPLAN Notices*, vol. 29, no. 8, pp. 1–12, 1994.

[13] Joshua S. Auerbach, Charles Barton, Mark Chu-Carroll, and Mukund Raghavachari, "Mockingbird: Flexible stub compilation from pairs of declarations," in *International Conference on Distributed Computing Systems*, 1999, pp. 393–402.

[14] Bryan Ford, Mike Hibler, and Jay Lepreau, "Using annotated interface definitions to optimize RPC," in *Symposium on Operating Systems Principles*, 1995, p. 232.

[15] Kris De Volder, "Aspect-oriented logic meta programming," in *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, Pierre Cointe, Ed. 1999, vol. 1616 of *Lecture Notes in Computer Science*, pp. 250–272, Springer Verlag.

[16] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig, "Stub-code performance is becoming important," in *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software (WIESS)*, Berkeley, CA, 2000, USENIX Association.