SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2225

Implementing an embedded compiler using program transformation rules

Tegawendé F. Bissyandé[1], Laurent Réveillère[2,*,†], Julia L. Lawall[3], Yérom-David Bromberg[2] and Gilles Muller[3]

[1]*SnT - University of Luxembourg, Luxembourg*
[2]*LaBRI - University of Bordeaux, France*
[3]*Inria Paris Rocquencourt, France*

SUMMARY

Domain-specific languages (DSLs) are well-recognized to ease programming and improve robustness for a specific domain, by providing high-level domain-specific notations and checks of domain-specific properties. The compiler of a DSL, however, is often difficult to develop and maintain, because of the need to define a specific treatment for a large and potentially increasing number of language constructs. To address this issue, we propose an approach for specifying a DSL compiler using control-flow sensitive concrete-syntax based matching rules. These rules either collect information about the source code to carry out checks or perform transformations to carry out compilation. Because rules only mention the relevant constructs, using their concrete syntax, and hide the complexity of control-flow graph traversal, it is easy to understand the purpose of each rule. Furthermore, new compilation steps can be added using only a small number of lines of code. We explore this approach in the context of the z2z DSL for network gateway development and show that it is beneficial to implement the core of its compiler in this manner. Copyright © 2013 John Wiley & Sons, Ltd.

Received 4 July 2012; Revised 13 August 2013; Accepted 13 August 2013

KEY WORDS: DSL; compiler construction; internal (embedded) languages; embedded compilers; program transformation

1. INTRODUCTION

Domain-specific languages (DSLs) expose concepts that are focused on a specific domain, via specialized syntax, thus providing many opportunities for improving developer productivity and empowering domain experts [1–5]. Designing and implementing such a language, however, raises practical challenges. Two main currents have emerged for these tasks: *external* DSLs and *internal* DSLs [6]. An external DSL is implemented using a standard compiler toolchain, consisting of a parser, code checker, if any, and code generator that are dedicated to the given DSL syntax. This approach provides maximum flexibility in the language design, because compilation tools can be specifically targeted towards the desired language features. However, this approach requires a high degree of compiler development expertise. An alternative is provided by an internal DSL, also known as an *embedded* DSL, where the language is implemented as a library, in terms of new operators and abstract data types, for some existing general-purpose host language. Common host languages include Haskell, Scala, and Ruby, which provide, to varying degrees, flexible syntax, advanced type systems, and reasonable performance that can support a variety of syntaxes and verifications, without the need to directly implement a parser, code checker, or code generator [7–10]. Internal DSLs, however, are limited to the kinds of syntax, verifications, and optimizations provided

*Correspondence to: Laurent Réveillère, LaBRI - University of Bordeaux, 33402 Talence, France.
†E-mail: reveillere@labri.fr

by the host language, and all implementation strategies target the requirements of the host language, not of the DSL itself.

In this paper, we present an approach to DSL implementation building on a case study in the context of the z2z DSL for network gateway development [4]. A *gateway* is the part of a network infrastructure that serves to translate messages between hosts that implement different network protocols. The environments in which we live are becoming more and more networked, and there is a corresponding explosion in network protocol designs, making the development of gateways increasingly necessary. Gateway code must perform complex message processing operations, taking into account nonfunctional variations between protocols, such as the choice of the underlying transport layer and the choice between synchronous and asynchronous messaging. Furthermore, to reduce costs, a gateway is typically implemented as an embedded system, for which careful coding is required to maximize performance and minimize resource usage. These issues complicate gateway development and imply that both efficiency and correctness are critical.

The z2z DSL is a C-like language that provides abstractions dedicated to the gateway domain, including protocol features, message structures, and the message translation logic. This language is complemented by a highly optimized runtime system, which manages the network interaction. The z2z DSL was originally designed and implemented as an external DSL following the *source to source transformation* pattern [11]. The implementation thus amounts to a *compiler* [12], in that it transforms code written in one language into another one. This compiler generates C code that can then be further compiled into executable code by using a standard optimizing C compiler. By passing through C, the implementation provides good performance and low memory usage. The z2z compiler furthermore performs a number of checks to ensure robustness. The z2z DSL has been used to implement a number of complex gateways, including a gateway to allow Session Initiation Protocol (SIP)[‡] control of a Real Time Streaming Protocol (RTSP)[§] camera, a service discovery gateway between Service Location Protocol (SLP)[¶] and Universal Plug and Play (UPnP),[‖] and a gateway to allow tunnelling Simple Mail Transfer Protocol (SMTP)[**] traffic over HyperText Transfer Protocol (HTTP)[††] [4]. We are also aware of some exploratory industrial usage of the language. A full description of the syntax and semantics of the z2z language can be found in previous work [13].

Our experience in designing and implementing z2z, as well as several other DSLs for systems programming tasks that require ad-hoc fine-grained code checks and aggressive optimization [14–17], has shown that the initial design of an external DSL and the implementation of its compiler can be done very quickly if developers are familiar with programming language concepts and compiler construction techniques. However, as in previous work by others [18], we have also found that a new DSL must frequently evolve, as new requirements and even new application areas become apparent. For example, in the development of the HTTP to SMTP gateway, we found that it was necessary for the message translation logic to explicitly manipulate Transmission Control Protocol (TCP) connections, a functionality that was not anticipated in the initial z2z design. Extending the z2z DSL to provide this functionality entailed the addition of new primitives, accompanied by the corresponding addition of new code checks. These changes to the z2z DSL required a pervasive transformation of its compiler. To support this kind of language evolution, the compiler design must make it possible to prototype code generation and code checking rules for new language features quickly, while keeping the language implementation understandable for later updates. If this flexibility cannot be provided, language developers and users are likely to abandon the DSL and revert to a general-purpose language-based solution.

The difficulties in implementing and maintaining the z2z compiler can essentially be attributed to the gaps, both syntactic and semantic, between the source language (z2z), the compiler implementation language (OCaml), and the target language (C). When a developer of the z2z

[‡]for multimedia sessions
[§]for control of streaming medias servers
[¶]for service discovery
[‖]for device-to-device networking
[**]for sending e-mails to servers
[††]for interacting with web servers

DOI: 10.1002/spe

language finds a problem in the generated code, he must track down the portion of compiler code that produces the defect and fix it. To do so, he must first correlate the generated C code with z2z code to determine which part of the z2z specification has been erroneously translated and then go through the internals of the compiler's OCaml code to fix the implementation. Whether such a fix involves modifying the checks performed by the compiler or improving the transformations carried out during the code generation process, it can entail significant side effects to the compiler design and may thus require reimplementing large portions of the compiler. Even if some parts that are affected are merely boilerplate code, the need to study and modify them, including the possibility of introducing errors, represents a substantial burden on the language developer.

These difficulties in maintaining the z2z compiler suggest that an internal DSL design and implementation approach may be more suitable in the long term. This requires the choice of a host language. The z2z user community is used to programming in C-like languages, and z2z critically relies on the performance that can only be achieved by a native C implementation. C, however, is not a natural target for defining internal DSLs, as it does not provide a rich type system or other DSL support features. Rather than relying on the C language to provide domain-specific features and enforce domain-specific properties, we propose an alternate approach to internal DSL development based on the use of externally developed transformation and code checking rules. For this, we use Coccinelle [19], a program matching and transformation tool for C code, to replace the compilation of z2z code to C. Coccinelle specifications are written using a syntax and following a structure very similar to that of the target C program itself, thus freeing the language developer from manipulating complex intermediate representations such as abstract syntax trees and control-flow graphs. Rules are furthermore modular, being specific to the affected constructs, making them simpler than a typical compiler pass. A thorough documentation on Coccinelle can be found on the project web page: http://coccinelle.lip6.fr.

The results of our work are

- We propose an approach to DSL compiler development that reduces the complexity of designing and maintaining the DSL compiler by expressing the required checks and transformations using a notation that is syntactically close to that of the generated code. To this end, we discuss our practical experience in first implementing the z2z DSL as an external DSL and later migrating it to an internal DSL. We motivate the need for migration and enumerate the benefits of the new approach.
- We describe how to use the Coccinelle program matching and transformation engine to perform the code checking and code generation steps required by an embedded compiler.
- We show the applicability of our approach by using it to implement an embedded compiler for the most complex and performance-demanding part of the z2z DSL, the translation of messages. The resulting compiler is about one third the size of the original compiler for the message translation code, which was implemented in OCaml.

The rest of this paper is organized as follows. Section 2 presents the z2z DSL and highlights the code checking and code generation steps that are performed by its existing compiler. Section 3 describes the changes required to the syntax of the z2z DSL to embed it into C. Section 4 presents Coccinelle and outlines our approach to internal DSL development. Section 5 assesses the efficiency of our approach, and Section 6 discusses related work. Finally, Section 7 concludes and presents future work.

2. A NETWORK PROTOCOL GATEWAY DOMAIN-SPECIFIC LANGUAGE

The z2z DSL provides facilities for defining a gateway in terms of three kinds of interdependent modules that describe *network protocol behaviors, message structures*, and the *message translation logic*, building on an optimized run-time system. A *protocol specification module* describes various properties of the interaction with the network, such as the transport protocol used, whether requests are sent in unicast or multicast, whether responses are received synchronously or asynchronously, and how to relate invocations that should be considered to be within the same *session*. It also specifies how to dispatch a received request to a specific message-translation handler for

DOI: 10.1002/spe

processing. A *message specification module* defines *message views* that describe the information that can be extracted from incoming messages and *templates* that describe the structure of new messages to be created. Both message views and templates are represented as a set of fields. Finally, the *message translation module* describes how to translate between the various types of messages, taking into account protocol properties.

The z2z DSL provides a sublanguage for describing each of these modules. Our language implementation case study focuses on the sublanguage message translation language (MTL) for implementing the message translation module. The message translation module describes actions to perform on each message, while the other kinds of modules primarily define supporting data structures and provide specifications. Thus, the message translation module is typically the most complex and performance-demanding part of a z2z gateway implementation. The compilation of this module, however, depends on information from the other modules.

Message translation language code consists of a set of handlers, one for each kind of relevant incoming request, as indicated by the protocol specification module of the source protocol. Domain-specific operators are provided for manipulating and constructing messages, for sending requests and returning responses, and for managing *sessions*, which maintain state across requests. Figure 1 illustrates an extract of the message translation module of an SLP to UPnP gateway. This handler takes SLP request *s* as an argument (line 1), constructs and sends a SSDP request to perform service discovery (line 6), and then constructs and sends an HTTP GET request to obtain contact information about the discovered services (lines 8–11). Note that the handler argument is not used in the service discovery process, because the goal is to become aware of all available services. The result is a message encapsulating the discovered URL. More details about MTL and the rest of the z2z DSL are presented in previous work [4].

The compiler of the z2z DSL, treating all three sublanguages, is implemented in OCaml and follows a traditional structure [12], consisting of a parser, which converts the source code to an abstract syntax tree (AST), followed by various compilation phases that analyze and transform the AST, and concludes with a code-generation phase, producing C code. This compiler performs some domain-specific code checks to detect inconsistent specifications and ensure the generation of safe code. We now describe the code checks and code generation steps that are performed by the compiler.

2.1. Code checks

The checking phase (1) performs consistency checks to ensure that the information declared in each module is used elsewhere according to its declaration and (2) applies dataflow analysis to the MTL code to ensure that values are well-defined when they are used.

Consistency checks. The consistency checks ensure that the MTL code is consistent with the protocol specification modules and the message specification modules. The protocol specification module of the source protocol declares how to dispatch incoming requests to the appropriate handlers and whether a response is expected from these handlers. The z2z compiler checks that the

```
1  slp response SrvReq (slp request s) {
2    ssdp response res_ssdp;
3    http request    req_http;
4    http response res_http;
5    // send a SSDP SEARCH request and receive a SSDP response
6    res_ssdp = send(Search(mx= 3, st="upnp:rootdevice"));
7    // send a HTTP GET request and receive a HTTP response
8    http_set_host(res_ssdp.location_ip,res_ssdp.location_port);
9    req_http = Get(path = res_ssdp.location_path, ip = res_ssdp.location_ip);
10   req_http.port = res_ssdp.location_port;
11   res_http = send(req_http);
12   return srvReply(url=res_http.URLBase);
13 }
```

Figure 1. Session Initiation Protocol (SLP) to Universal Plug and Play (UPnP) gateway (message translation language specification extract).

DOI: 10.1002/spe

message translation module defines a handler for each kind of expected request, as indicated by the protocol specification module, and that this handler has the expected return type. A message specification module indicates the set of fields associated with each message view and template and their types. A message specification module furthermore indicates which fields are automatically handled by the runtime system (fields declared as `private`) and which must be managed by the message translation module (fields declared as `public`). The z2z compiler checks that fields are only used in the allowed module and that every access or update has the declared type.

Dataflow analysis. The z2z compiler uses dataflow analysis to check the safety of MTL operations on messages, to ensure that fields are initialized before they are accessed, and to ensure that message structures are fully initialized before they are transmitted on the network. The compiler also uses dataflow analysis to ensure the correct use of operations related to the management of sessions and TCP connections.

An MTL handler, as illustrated in Figure 1, is parameterized by a *view* of the corresponding request (line 1), organized as defined in the message specification module. The information in this view can be extracted using the standard structure field access notation. If the message specification module indicates that a view element is optional, the z2z compiler checks that any reference to the corresponding field is preceded by an `empty` test to determine whether its value is available before it is used.

To create a message, an MTL handler invokes the name of the corresponding template (lines 6, 9, and 12 in Figure 1), as defined in the message specification module, and optionally passing to it some keyword arguments indicating the values of some or all of the fields. The remaining fields may then be incrementally initialized (line 10) until the created message is flushed to the network at the point of a `send` or `return` operation (lines 6, 11, and 12). Template fields may be declared in the message specification module to be `public` or `private`. The z2z compiler checks that all public fields of a template are initialized before the template is passed to `send` or `return`.

Finally, the dataflow analysis also checks properties of sessions and of TCP connections. If the protocol specification module for the source protocol of the gateway declares that this protocol is session-based, then the message translation module may declare session variables. Such variables are defined outside of any handler and keep their values across successive requests. The z2z compiler checks that references to session variables do not occur outside session boundaries, as defined by the `session_start` and `session_end` operations. Similarly, if the protocol specification module for a target protocol of the gateway declares that the protocol relies on TCP, then the message translation module may use some operations (`tcp_connect`, `tcp_get_connection`, etc.) to explicitly manage the TCP connection. The z2z compiler performs various checks to detect erroneous uses of these operations. For example, only `tcp_connect` can be used after a call to `tcp_close` along a given control-flow path.

2.2. Code generation

The code generation phase generates C code. We focus on the generation of C code from an MTL specification. For that purpose, the code generation phase carries out three main categories of transformations: (1) implementation of asynchronous message sends; (2) implementation of the variables used by MTL; and (3) implementation of memory management.

The send operation. A request is sent from the gateway to the target service using the operator `send`, as illustrated in lines 6 and 11 of Figure 1. If the target protocol specifies that responses are returned asynchronously, then the gateway must be allowed to perform other tasks until the response is available. So that the run-time system can restart the handler, the compiler generates code to pass `send` a pointer to the remainder of the current handler, amounting to a *continuation* [20, 21].

Variables. Local variables that must be maintained across asynchronous `send`s are identified and implemented as elements of an environment structure. Similarly, session variables, which must be maintained across multiple handler invocations, are implemented in a global environment.

DOI: 10.1002/spe

Dynamic memory management. When memory needs to be allocated dynamically, as in an invocation of a template constructor for message creation, the z2z compiler generates code to manage reference counts. These reference counts are used to ensure proper memory management, without introducing the run-time overhead of a tracing garbage collector, as found in languages such as Java.

2.3. An increasingly obsolete compiler

Over the course of the development of z2z, the syntax of MTL, its semantics as well as the internals of its compiler have all been refined several times. Changes have included fixing bugs in the compiler and extending the language to support new features requested by users. As the compiler has evolved, we have found that adding new features or modifying existing ones was requiring an increasing amount of effort. For instance, when adding support for TCP-based networking, we had to make changes pervasively throughout our manually written compiler; this had the effect of introducing bugs that were tedious to find and fix. Indeed, out of the 26 OCaml implementation files (.ml) that constitute the z2z compiler, two were added and 17 were modified, over the course of 2 days. These files amount to over 80% of the MTL compiler code. Twenty other edits in all the modified files were necessary to fix bugs in the implementation of the TCP operators in the following 2 weeks. In the long term, the maintenance that would be necessary to extend the language to address domain changes (e.g., new protocol paradigms) and developer needs for complex applications could, in practice, become impossible.

An approach that achieves part of the effect of language extension is to make it possible to link programs with externally developed libraries. For instance, the use of encryption libraries is now commonplace for extending protocols to support encrypted communication and secure identification [22–24]. Nevertheless, the emphasis on static code checks and safety in the design of the z2z DSL means that MTL code cannot simply directly call existing libraries. An alternative would be to add these libraries to the run-time system, but doing so would require substantial expertise in the design of the run-time system. An *internal DSL*-based approach, in which checks are targeted to the DSL code, while unchecked host language code is supported as well, potentially provides a way around this problem. We consider implementing MTL in this manner.

3. MESSAGE TRANSLATION LANGUAGE AS A C INTERNAL LANGUAGE

Our approach to implementing a DSL as an internal language in C involves expressing the language in a C-like syntax and then using the program transformation tool Coccinelle [19] to transform the language constructs into C code that expresses their semantics. To this end, we first reorganize the syntax of MTL so that it is recognizable as valid C code by Coccinelle. In the C language, syntax extensions are restricted to what can be expressed using preprocessor macros, which are limited to single identifiers or a function-call-like notation; it is not possible to, for example, define infix operators, as permitted in languages such as Haskell and Scala and as are frequently used in DSL implementations [11, 25, 26]. Fortunately, as illustrated in Figure 1, the syntax of MTL is already quite close to that of C. We have found that changes are only required to the syntax of message-variable declarations and `foreach` loops to produce our C-MTL language. We do not believe that either of these changes has a significant impact on the understandability of the language.

Message-variable declarations. As illustrated in lines 1–4 of Figure 1, MTL declarations related to messages have the form of a double type declaration, containing both an indication of the protocol associated with the message and an indication of whether the message is a request or response. For example, in the handler header (line 1), the return type indicates both the name of the source protocol and whether a response is required (`response` or `void`), and the parameter type indicates again the name of the source protocol and that the parameter represents a request. Furthermore, the variable *req_http* on line 3 represents an HTTP request, while the variable *res_ssdp* on line 2 represents a SSDP response.

C variable declarations include only one type, and thus, these double type annotations found in MTL code do not represent valid C syntax. We thus reorganize these declarations. For the handler

DOI: 10.1002/spe

```
1  response SrvReq (slp s) {
2    DECLARE_RESPONSE(ssdp, res_ssdp);
3    DECLARE_REQUEST(http, req_http);
4    DECLARE_RESPONSE(http, res_http);
5    // send a SSDP SEARCH request and receive a SSDP response
6    res_ssdp = send(Search(mx= 3, st="upnp:rootdevice"));
7    // send a HTTP GET request and receive a HTTP response
8    http_set_host(res_ssdp.location_ip,res_ssdp.location_port);
9    req_http = Get(path=res_ssdp.location_path, ip = res_ssdp.location_ip);
10   req_http.port = res_ssdp.location_port;
11   res_http = send(req_http);
12   return srvReply(url=res_http.URLBase);
13 }
```

Figure 2. The Session Initiated Protocol (SLP) to Universal Plug and Play (UPnP) gateway of Figure 1 in C-message translation language (MTL) syntax.

header, the protocol name in the return type and the keyword `request` in the declaration of the parameter are redundant, as the same protocol is mentioned in both, and the parameter always represents a request message. In these cases, we simply drop the redundant information, as illustrated in line 1 of Figure 2. Local variables, on the other hand, can represent either requests or responses and can be associated with any protocol. For such variables, both the designation as `request` or `response` and the name of the protocol are thus essential. In this case, we have followed a strategy used in Linux kernel code and defined macros `DECLARE_REQUEST` and `DECLARE_RESPONSE` to declare these variables, as illustrated on lines 2–4 of Figure 2. These macros take as arguments the name of the protocol and the variable name, making it possible to preserve information about both the kind of message and the associated protocol, for use in the code checking and code generation process. MTL also allows the declarations of variables representing simple messages, which are not oriented as either requests or responses, and of lists of various types. For declaring such variables, C-MTL provides the macros `DECLARE_MESSAGE` and `DECLARE_LIST`, defined similarly.

Foreach. The MTL provides a `foreach` loop construct, in which the loop header declares a loop index variable using a local declaration as in C++, and expresses the possible values of this variable by assigning it to the list over which iteration is required. An example of such a loop is as follows, taken from the z2z specification of a SIP to RTSP gateway developed in our previous work [4].

```
foreach (fragment rtsp_m_ = rtsp_medias) { ... }
```

`foreach` is not part of C, and thus, it would normally be necessary to define such a construct as a macro. The C parser of Coccinelle, however, as part of its effort to parse C code without expanding macros [19], already recognizes `foreach` as a loop construct and permits a variable declaration in the loop header. Coccinelle, however, requires that the declaration end with a semicolon, as in an ordinary C statement, and thus, C-MTL requires this semicolon as well.

Note that although the C-MTL `foreach` essentially has the form of C code, it does not follow the C semantics, in that `rtsp_m_` is intended to be assigned to each of the elements of `rtsp_medias`, rather than to the complete list itself. A similar use of C syntax, without the associated C semantics, is illustrated in the use of keyword arguments in the instantiation of templates, as found on line 6 of Figure 2. There, the initializations are intended to be to the fields of the template, not to the individual variables. Thus, although we use a C-based syntax for C-MTL, we reserve the right, via the compiler, to assign to the various constructs domain-specific semantics. The C-MTL user should thus keep in mind the MTL semantics and not that of C.

4. A COCCINELLE-BASED EMBEDDED COMPILER FOR C-MESSAGE TRANSLATION LANGUAGE

Although the syntax of C-MTL is recognizable as C code, the domain-specific constructs, related to the management of network protocol messages, have a z2z-specific semantics. Thus, to obtain an executable gateway, the C-MTL embedded compiler must transform the domain-specific constructs into C code that implements their intended semantics. Furthermore, it is desirable to restrict the

DOI: 10.1002/spe

ordinary C code found in the C-MTL handlers, to protect the integrity of the message processing. The C-MTL embedded compiler must thus also enforce the intended properties. In this section, we show how these tasks can be carried out using the program-transformation tool Coccinelle.

In the rest of this section, we first give an overview of Coccinelle and then illustrate its use in defining the main steps of the compilation process: (1) code preprocessing, to simplify and highlight some aspects of the code; (2) code checking; and (3) code generation, as shown in Figure 3.

4.1. Coccinelle overview

Coccinelle is a tool for performing control-flow-based program searches and transformations in C code [19]. It provides a language, semantic patch language (SmPL), for specifying searches and transformations and an engine for performing them. An SmPL *semantic patch* consists of a sequence of *rules* based on pattern matching but can contain OCaml or Python code, to be able to perform arbitrary computations. Coccinelle allows code to be matched and transformed according to patterns that are very similar to the affected code itself. This property helps make the rules easy for the compiler maintainer to read. Furthermore, each rule only mentions the specific kinds of code fragments that are relevant to the associated compilation step. Thus, changes in the semantics of a construct of the DSL typically have only a localized impact on the compiler structure. This property further eases compiler maintenance.

We first present an overview of the SmPL syntax, via some simple semantic patches that perform checks and transformations required by the C-MTL compiler. A complete grammar of SmPL is available on the Coccinelle web site (http://coccinelle.lip6.fr/).

Example 1: Restricting the use of C code. Figure 4 shows a semantic patch to prevent the use of `setjmp` and `longjmp` in MTL handler code. The use of `setjmp` and `longjmp` in handler code can interfere with the compiler's ability to analyze the handler control-flow and thus could impair

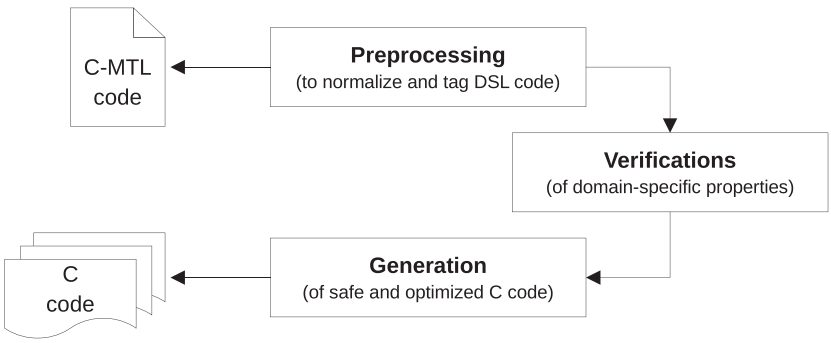

Figure 3. C-message translation language compilation steps.

```
1  @cmtl_handlers@
2  identifier handler, proto;
3  declarer name DECLARE_HANDLER;
4  @@
5  DECLARE_HANDLER(proto, handler);
6
7  @hasjmps exists@
8  identifier cmtl_handlers.handler;
9  position pos;
10 @@
11 handler (...) {
12 ... when any
13 (
14 setjmp@pos (...)
15 |
16 longjmp@pos (...)
17 )
18 ... when any
19 }
20
21 @script:ocaml@
22 h << cmtl_handlers.handler;
23 p << hasjmps.pos;
24 @@
25 let p_ = List.hd p in
26 let f = p_.file in
27 let l = p_.line in
28 Printf.eprintf "%s:%d in %s:
29  error : setjmp/longjmp
30        forbidden\n" f l h
```

Figure 4. A semantic patch for reporting all uses of `setjmp` and `longjmp` in C-message translation language handlers.

DOI: 10.1002/spe

its ability to perform the necessary checks of the message processing code. Furthermore, we expect that these operators are not typically useful in translating network protocol messages. The semantic patch consists of three rules: an SmPL rule named cmtl_handlers for collecting the names of the relevant protocol handlers (lines 1–5), an SmPL rule named hasjmps matching calls to `setjmp` or `longjmp` in the body of one of these handlers (lines 7–19), and an OCaml rule generating an error message when such a call is found (lines 21–30).

As illustrated by the first rule cmtl_handlers (lines 1–5), an SmPL rule begins by declaring some metavariables and then uses these metavariables to describe a pattern to be matched in the source code. cmtl_handlers defines two metavariables (line 2): *handler* that will contain the name of a handler and *proto* that will contain the name of its corresponding protocol. cmtl_handlers also declares that `DECLARE_HANDLER` is a variable-declarer name, as introduced in Section 3. The pattern (line 5) then instantiates these metavariables according to the arguments of the various `DECLARE_HANDLER` declarations. These `DECLARE_HANDLER` declarations are generated by a separate compiler, described in Section 4.2, from the protocol specification module of the gateway's source protocol.

The second rule, hasjmps, declares two metavariables: *handler* and *pos*. The metavariable *handler* is inherited from cmtl_handlers, implying that hasjmps is applied once for each possible binding of *handler*. The metavariable *pos* is new in hasjmps and will be used to store the position of each call to `setjmp` or `longjmp` in the source code. The pattern, on lines 13–17, then matches either a call to `setjmp` or a call to `longjmp`, as indicated by the use of a *disjunction*, represented by (, |, and) in the leftmost column. The calls to `setjmp` and `longjmp` can have arbitrary arguments, as indicated by the pattern '`...`' in their argument lists. The pattern '`...`' (lines 12 and 18) is also used before and after the calls to `setjmp` and `longjmp` to indicate an arbitrary sequence of statements. The annotation `when any` on the pattern '`...`' indicates that there are no constraints on the statements that can appear in these sequences. Finally, the notation `@pos` records the position of each occurrence of the keyword `setjmp` or `longjmp` in the position variable *pos*.

The final rule is an OCaml rule (lines 21–30). Such a rule also begins by defining some variables and then uses the variables in arbitrary OCaml code (lines 25–30). In this case, the OCaml rule inherits the metavariable *handler* from the SmPL rule cmtl_handlers and the metavariable *pos* from the SmPL rule hasjmps. Because hasjmps, which declares *pos*, inherits *handler*, the OCaml rule is applied once for each observed pair of bindings of these variables; on the other hand, if the metavariables had been defined disjointly, the OCaml rule would be applied once for each element of the cross product of their values. After binding the metavariables, the OCaml code prints an error message indicating the position at which the call to `setjmp` or `longjmp` occurs. In this work, we only use OCaml rules for generating error messages in the checking rules. OCaml is not involved in any of the code manipulation tasks.

The rules are applied to the C-MTL code in sequence, with the first rule being applied to each top-level term (variable declaration or function definition), then the second rule being applied once to each top-level term for each assignment of its inherited metavariables, etc. There are no loops in this process, and thus, it is guaranteed to terminate.

Example 2: transforming C-message translation language code into C code. Figure 5 shows two semantic patch rules related to injecting the MTL semantics of strings into C code. Unlike C strings,

```
1  @@
2  constant char [] C;
3  expression E;
4  @@
5  (
6  - E == C
7  + strcmp(E, C) == 0
8  |
9  - E != C
10 + strcmp(E, C) != 0
11 )
12 @@
13 constant char [] C; identifier handler;
14 fresh identifier str = "_mtl_string_";
15 @@
16 ++static struct string str = { -1, C };
17 handler(...) {
18 <+...
19 - C
20 + str.val
21 ...+>
22 }
```

Figure 5. A semantic patch implementing the z2z semantics of string constants.

DOI: 10.1002/spe

C-MTL strings can be compared using `==` and `!=`, which compare their content rather than their pointers, and they are represented as structures containing a reference count and the string value. The first rule (lines 1–11) rewrites comparison expressions involving constant strings to use `strcmp`. The SmPL operator `-` at the beginning of lines 6 and 9 instructs Coccinelle to remove the matched expressions, while the SmPL operator `+` at the beginning of lines 7 and 10 instructs Coccinelle to add the corresponding code to the generated program. The added code is constructed using the metavariable bindings obtained by matching the removed code, as well as by matching any pattern that may be present without `-/+` annotations. This pattern should be embedded in the declaration of a handler, as was done in the rule hasjmps in Figure 4, to ensure that the rule only applies to MTL code. We omit this detail to simplify the presentation.

The second rule (lines 12–22) replaces each constant string by a static structure containing the required reference-count information. For this, the rule matches each handler and each of the constant strings *C* occurring anywhere inside it, as indicated by the *nest* operator $< +... \ ...+ >$ (lines 18-21). For each constant string, we need to construct a corresponding structure before the beginning of the handler and give this structure a unique name. Thus, a fresh identifier *str* is declared for each constant string *C*, based on the prefix *_mtl_string_* (line 14), and a corresponding declaration of a structure, named *str*, is added before the definition of the handler. The annotation $++$ (line 16) allows these added structure declarations to accumulate. We also need to replace each occurrence of the string by an access to the appropriate structure field. Thus, in the nest, the rule replaces each matched occurrence of *C* by an access to the `val` field of the corresponding fresh *str* variable.

As in the previous example, the first rule (lines $1-11$) is applied once to each top-level term in the program, and then, the second rule is applied once to each top-level term in the transformed result produced by the first rule. Thus, for example, each constant string involved in a string comparison operation that is created by the first rule is replaced by a reference-counted structure and appropriate field accesses, by the second rule. It is also necessary to update the string references stored in variables to refer to the structure field. This is done by a subsequent rule, which is not shown.

4.2. Preprocessing

We now turn to the implementation of the Coccinelle-based compiler itself, starting with a preprocessing phase. To facilitate code checking and code generation using Coccinelle, the compiler first reorganizes the MTL code, to make some operations explicit and to add tags that indicate important points in the code. This step mainly serves to avoid redundant processing in the subsequent code checking and code generation steps.

Incorporating information about protocols and message. The code checking and code generation steps require information from the protocol specification and message specification modules. To make this information available, we have implemented a dedicated compiler that encodes the contents of these files into C structure declarations in header files that exist only for the compilation process. The use of a dedicated compiler in this case seems essential, because these modules are declarative, and thus are not naturally expressed as C code, which is required for processing using Coccinelle. These modules serve as a critical reference for the gateway programmer and thus must be easily understandable.

The dedicated compiler uses auxiliary information about the set of protocols involved in the gateway to add `includes` of the generated header files to the top of the C-MTL code. This phase also enhances the declarations of local variables declared using `DECLARE_REQUEST` or `DECLARE_RESPONSE`, to distinguish between a received message, which has the type of the corresponding message view, and a constructed message, which has `template` type.

Normalization of message construction code. In MTL, a message is constructed by invoking a template on a set of keyword arguments that provide values for any subset of the fields of the message template. The remaining fields can then be initialized by explicit assignments. Coccinelle rules are used to normalize the message construction code, by removing any

DOI: 10.1002/spe

```
1  @prot_info@ identifier prot; @@
2  struct protocol_info prot = {
3    .place = TARGET,
4    .sync = ASYNC,
5  };
6
7  @async exists@
8  declarer name DECLARE_REQUEST;
9  identifier handler, prot_info.prot, req;
10 fresh identifier pr=prot##"_protocol";
11 @@
12 handler(...) {<+...
13 DECLARE_REQUEST(prot,
14                 template,req);
15 ... when any
16 - send(req)
17 + send_async(&pr,req)
18 ...+>
19 }
20
21 @sasync@
22 position p;
23 @@
24 send_async@p(...)
25 @script:ocaml async_label@
26 p << sasync.p;
27 label;
28 @@
29 label := Printf.sprintf "cont_%d_%d"
30     (List.hd p).line (List.hd p).col
31
32 @insert_label@
33 position sasync.p;
34 expression protocol,arg,env, res;
35 identifier async_label.label, async.f;
36 @@
37 f(...) {
38 ... when any
39 -res =
40 send_async@p(protocol,arg,env
41 + ,&&label
42  );
43 + return NULL;
44 + label:
45 + SEND_CONT();
46 + res = get_response();
47 ... when any
48 }
```

Figure 6. Inserting labels for continuations.

template arguments and replacing the call to the template by a call to the message-creation operator `new` having the template name as an argument. The original message arguments become assignments to the fields of the allocated message. A local request or response variable is introduced to hold the message, according to the role of the protocol, as indicated by the type of the template.

Making control-flow explicit. A `send` to a protocol that responds asynchronously does not directly return a result but instead aborts the execution of the handler until the response is available. The implementation of such an asynchronous `send` must thus provide to the run-time system information about what code should be executed at that time. For this, Coccinelle rules are used to construct a form of *continuation*, represented as a label, which is passed to the call to `send` and renamed `send_async` to indicate its specific semantics. The handler is then reorganized such that it takes a label or `NULL` as a parameter and, if a label is provided, jumps to the corresponding position in the handler.

Figure 6 shows extracts of the semantic patch for managing asynchronous `send`s. The rule prot_info (lines 1–5) uses the header files generated from the protocol specification module to identify the target protocols of the gateway for which `send`s are asynchronous. For each such protocol, the rule async (lines 7–19) transforms a `send` (line 16) whose argument is a variable declared to be a request for the given protocol (lines 13–14) into a call to `send_async`. Information about the protocol is also provided to `send_async` as the first argument. Then, for each call to `send_async` (line 24), an OCaml script (lines 25–30) creates a name for a new label, based on the position of the call in the MTL code (lines 29–30), obtained using the rule sasync (lines 21–24). Finally, the rule insert_label adds the address of this label as an argument to `send_async` and places this label after the call to `send_async` (line 44), taking care to halt the handler first with a `return` statement (line 43). A tag `SEND_CONT` is also inserted just after the label (line 45) to indicate that this is the entry point of a continuation that comes after a `send` operation. Another tag that does not appear in the example is `START_CONT`, which indicates the entry point of the handler.

4.3. Code checks

As summarized in Section 2.1, the goal of the checks of the message translation module is to ensure that this module is consistent with the protocol and message specification modules and to ensure that values are well-defined before they are used. These checks rely on the information in the header files that were included as part of the preprocessing phase.

DOI: 10.1002/spe

Consistency checks. As introduced in Section 2, template fields may be annotated as `public` or `private`, and the MTL code is only allowed to access the `public` fields. The semantic patch shown in Figure 7 detects attempts to initialize private fields. The first rule, ref (lines 1–5), matches a structure field initialization, recording the name of the structure type in the metavariable *T* and the name of the accessed field in the metavariable *fld*. The second rule, priv (lines 7–15), then matches the structure type, included via a header file, checking for a declaration of *fld*. In this structure, *fld* is represented as having a function type, in which the parameter types indicate the properties of the field. This function type is not used in the execution of the gateway; instead, it is used as a means to collect the various attributes of each template field, as indicated in the corresponding message specification module. The rule priv checks whether the type of the second 'parameter' of this function type is `private` (line 13). The types of the other parameters of this function indicate whether the field is mandatory or optional (*mand_or_opt*) and whether it is preinitialized with a default value (*init_or_uninit*). Finally, an OCaml script (lines 17–22) prints an error message in the case where a match of the ref rule also satisfies the priv rule.

The main advantage of the semantic patch described previously is that it only mentions the specific kinds of code that are relevant to the property to be checked. This makes the check understandable for any programmer with basic knowledge of C programming and SmPL. Writing consistency checks in this manner makes the language implementation understandable, which in turn makes maintenance easier.

Dataflow analysis. Checks performed by the z2z compiler also include dataflow analysis to ensure the validity of the generated gateway code. For example, a gateway should not be allowed to send network message packets with uninitialized fields. To check this property, the compiler must be able to reason in terms of paths in a control-flow graph. Coccinelle provides this capability via the '...' operator.

In the semantic patch of Figure 8, the rule public_fld collects all the `public` message fields (line 7) associated with each template. On the basis of this information for a given template, the rule sending then checks, for each public field, whether there exists (keyword `exists`, on line 11) a path in the control-flow graph from the call to `new` to a call to `send` that does not include either a reinitialization of the variable holding the result of `new` or an initialization of the public field. In this rule, '...' (line 17) represents the sequence of operations between `new` and `send`, and the

```
1  @ref@
2  type T; position pos; expression E;
3  idexpression T x; identifier fld;
4  @@
5  x@pos->fld = E
6
7  @priv@
8  type ref.T, T1, mand_or_opt, init_or_uninit;
9  identifier ref.fld; typedef private;
10 @@
11 T { ...
13   T1  (*fld)(mand_or_opt, private, init_or_uninit);
14   ...
15 };
16
17 @script:ocaml depends on priv@
18 x << ref.x; p << ref.pos; fld << ref.fld;
19 @@
20 let line = (List.hd p).line in
21 Printf.printf "error: %d : field %s
22     of %s is private\n" line x fld
```

Figure 7. Consistency check: a message translation language specification should not update `private` fields of a message template.

```
1  @public_fld@
2  type T; t; identifier tm,fld;
3  typedef public,uninitialized;
4  @@
5  struct tm {
6    ...
7    T (*fld)(t,public,uninitialized);
8    ...
9  };
11 @sending exists@
12 position p0, p1;
13 expression E, x, y;
14 identifier public_fld.tm, public_fld.fld;
15 @@
16 x@p0 = new(tm);
17 ... when != x = E
18     when != x.fld = E
19 send@p1(x);
```

Figure 8. Dataflow analysis: all `public` fields of a template instance must be initialized before the instance is passed to `send`.

DOI: 10.1002/spe

```
1  @needs_locks@
2  identifier globals.i, handler;
3  @@
4  handler(...) { <+... i ...+> }
5
6  @locks depends on !bad_session@
7  identifier needs_locks.handler;
8  @@
9  handler (...) {
10 <...
11 (
12   session_start();
13 + lock();
14 |
15   \(START_CONT\|SEND_CONT\)();
16   ...
17   session_start();
18 |
19   \(START_CONT\|SEND_CONT\)();
20 + lock();
21 )
22 ...>}
25 @unlock1@
26 position bad_return;
27 @@
28
29 + unlock();
30   session_end();
31   ...
32   return@bad_return ...;
33
34 @unlock2@
35 position p != unlock1.bad_return;
36 @@
37
38   lock();
39   ... when strict
40 + unlock();
41   return@p ...;
```

Figure 9. Code generation: in continuations that start sessions, the lock is acquired after a session is started.

`when !=` operators describe terms that should not appear within this sequence (lines 17 and 18). An OCaml script, which is not shown, is then used to print an error message if sending is satisfied.

4.4. Code generation

In a traditional AST-based compiler, it is necessary to generate code for all of the constructs of the language. In our approach, code generation is only needed for terms whose C semantics is different from the semantics that is desired for the DSL construct. In the case of the z2z DSL, the main issues that require code generation are the management of session variables, of local variables that are live across asynchronous `send`s, of memory (as illustrated by the introduction of reference count management in Section 4.1), and of the `send` operation.

Variables. To implement sessions, the values of session variables must be maintained across invocations of the handlers. Session variables are represented in C-MTL as global variables, but they cannot be implemented in this way, because a session variable is specific to each initiated session and not global to all invocations of the handlers. To reduce the scope of the session variables, we reimplement them as fields of an environment structure. This structure is created when a session is started and is then passed by the run-time system to subsequent handler invocations that are within the same session, as defined by the criteria specified in the protocol specification module of the source protocol. Local variables that are live across a call to asynchronous `send`s also must be maintained across successive calls to the same handler, as the responses for the `send`s become available. The compiler introduces a similar environment structure in this case.

An additional transformation is needed in the case of session variables. Because the run-time system is multi-threaded, multiple handlers within the same session can be invoked in parallel. Thus, any handler that refers to a session variable should acquire a lock associated with the session before using the session variables. If a lock is not held when needed, the result can be an inconsistent access to the session variables, and if a lock is not released when it should be, the result can be deadlock.

Figure 9 shows an extract of the semantic patch that transforms handlers that process global variables, identified using the rule needs_locks, to insert lock acquisitions after `session_start` (lines 12–13) and at the beginning of all continuations that do not contain `session_start` (lines 19–20) and adds lock releases before a `session_end` and at the end of continuations that do not contain `session_end` (rules unlock1 and unlock2).[‡‡] These rules rely on the rule globals, which is not shown, that collects the names of the session variables, storing each in the metavariable

[‡‡]Backslashes are used in a disjunction that appears within a single line.

DOI: 10.1002/spe

```
1  @@
2  identifier t, res, tmpl.templ;
3  fresh identifier i = "i";
4  fresh identifier _r = "resp";
5  fresh identifier view_new = "msg_"
6      ## tmpl.proto ## "_view_new";
7  fresh identifier pr = tmpl.proto
8                    ## "_protocol";
9  fresh identifier parser = tmpl.proto
10                         ## "_parser";
11 @@
```

```
11  t = new(templ);
12  ...
13  - res = send_sync(t);
14  + template_sending_req(t);
15  + {
16  +   size_t i; char *_t; char *_r;
17  +   _t = template_flush(t, &i);
18  +   res = view_new();
19  +   _r = resp_send_sync(&pr,_t,i);
20  +   parser(strlen(_r),_r,res);
21  + }
```

Figure 10. Code generation: the high-level synchronous-`send` primitive is translated into various actions for sending messages, waiting for responses, and parsing responses for future manipulations.

i, and a rule bad_session that is used to ensure that the transformations are only performed on continuations where `session_start` and `session_end` are correctly used.

The `send` *operation.* In MTL, a `send` operation is expressed as a function call taking a template as an argument and possibly returning a message view as a result. In Figure 6, we have seen part of the implementation of an asynchronous `send` and its effect on the handler control flow. Here, we focus on the process of constructing and sending the messages and processing the result for synchronous sends.

Figure 10 shows the semantic patch that generates code for sending requests synchronously. This semantic patch replaces the call to `send` by a series of more primitive operations. First, the private fields of the template are initialized (`template_sending_req`, line 14), and then, the resulting template is flushed to a string (line 17) that can be sent on the network (line 19). Finally, the raw data that is received as a response is parsed to fill a message view, `res`, (line 20) that can be processed as an ordinary structure in the remainder of the program. Several of these operations, such as constructing the view of the response and parsing the raw response data to fill this view, rely on functions specific to the protocol associated with the sent message. Fresh identifiers (lines 3–10) are used to create names to be used to store temporary values (lines 3–4) or to reference these protocol-specific operations (lines 5–10).

5. EVALUATION

We compare our approach with our previous work in which we developed a traditional, AST-based compiler from the z2z DSL to C using OCaml [4]. We have used the z2z DSL to specify a number of gateways: between SIP and RTSP, between SLP and UPnP, and between SMTP and SMTP via HTTP. In each case, both compilers produce gateways that are equivalent in terms of executable binary size and performance. Therefore, we focus our evaluation on the size of the compiler and the compiler execution time. We also illustrate the process of correcting a bug in the Coccinelle-based compiler.

Code size. The size of the compiler gives an idea of the difficulty that may confront a language maintainer. Table I shows that our original MTL compiler contains over three times as many lines of code as the Coccinelle-based one. Maintenance of the Coccinelle-based C-MTL compiler is also eased by the fact that it is clearly separated into preprocessing, checking, and generation phases; in the original MTL compiler, these issues are somewhat mixed, making it hard for a maintainer to separately address issues in one part without introducing and needing to deal with side effects in the others. Finally, the fact that SmPL matching and transformation rules are syntactically close to the original code and the generated code eases the maintenance of the language implementation.

Table II shows in more detail the size of the different parts of the compiler that deal with specific language features. For instance, for the *send* operation, we consider the transformation rules involved in the code tagging process for managing continuations and in generating the appropriate

DOI: 10.1002/spe

Table I. Code size, in lines of code (LOC), of the C-message translation language (MTL) and MTL compilers.

		Compiler code size (LOC)	
Coccinelle-based C-MTL compiler	1850	529	code reorganization
		403	code checks
		918	code generation
OCaml-based MTL compiler	6196		

Table II. C-message translation language compiler code size for specific language features.

Compiler feature		# SmPL rules	C-MTL compiler code size (LOC)
Consistency checks	Message view fields – mandatory, public, and read-only	6	189
	Message template fields – mandatory, public, and initialized	6	
Dataflow analysis	Initialization of public fields – empty test prior to access	4	108
	Initialization of message structures – before send	5	
Message construction	Convert message constructors	5	355
	Fill message templates and fields	13	
Support for TCP connections	Add TCP protocol support	7	110
	Add code for opening/closing TCP connections	2	
Session management	Processing of session variables	6	160
	Handling of continuations	6	
	Insertion of locks for session variables	17	
Send operation	Rewrite *return* statements	10	388
	Rewrite TCP response handlers	7	
Miscellaneous	Add parameters to handlers	9	173
	Add headers for runtime system primitives	6	
	Modify connection functions names	6	
	Normalize string comparisons (with strcmp)	1	

SmPL: semantic patch language; MTL: message translation language; LOC: lines of code; TCP: Transmission Control Protocol.

code for different protocol behaviors. In this case, the limited number of lines of code combined with the simplicity of the transformations simplifies debugging.

Whereas the MTL compiler has to generate code for all of the constructs of the language, the C-MTL compiler performs code generation only for terms whose C semantics is different from the C-MTL semantics, which drastically reduces the size of the compiler. Indeed, like any internal DSL, C-MTL reuses many features of the C language, such as arithmetic operators, for free, without requiring any extra development effort.

Compilation time. To measure the compilation time, we use a Dell 2.40 GHz Intel ® Core ™ 2 Duo with 3.9 GB of RAM, running Ubuntu with Linux kernel 2.6.32. We use Coccinelle 1.0.0-rc1, compiled using OCaml 3.11.2. We test both compilers on a gateway between the SLP and UPnP service discovery protocols, of which an excerpt was shown in Figure 1.

The original MTL compiler requires only 0.101 seconds to compile the SLP-UPnP gateway, while the Coccinelle-based C-MTL compiler requires 4.192 s. Indeed, just the code generation for the send

DOI: 10.1002/spe

```
1  @needs_locks@
2  identifier cmtl_handlers.handler;
3  identifier globals.i;
4  @@
5  handler(...) {
6   <+... i ...+>
7  }
8
9  @buggy depends on needs_locks@
10 identifier cmtl_handlers.handler;
11 @@
12 handler(...) {
13 <...
14 (
15    START_CONT ();
16 + lock();
17 |
18    SEND_CONT();
19 + lock();
20 )
21 ...>
22 }
23
24 @unlock@ @@
25    lock();
26    ... when strict
27 + unlock();
28    return@p ...;
```

Figure 11. Code generation: handlers that refer to the global environment need locks at the beginning of all of their continuations.

operation, as described previously, requires 0.212 s, which is more than the total time of the original MTL compiler on the entire gateway. As opposed to the original MTL compiler, which is dedicated to that purpose, the C-MTL compiler relies on Coccinelle, which is a general-purpose program matching and transformation engine for C. Furthermore, to compile a C-MTL specification, we launch Coccinelle successively for each semantic patch, requiring the C-MTL code to be read from disk and parsed at each step, whereas the MTL compiler parses the code only once before carrying out, at once, all checking and generation steps. It should be straightforward to modify Coccinelle to address this issue.

Correcting a buggy compilation rule. In Figure 9, we showed how to add the use of locks to protect accesses to session variables. Our first, incorrect, attempt at implementing this functionality is shown in Figure 11. This semantic patch identifies handlers that reference global variables (rule needs_locks), inserts lock acquisitions at the beginning of all continuations in such handlers (rule buggy), and adds lock releases at the end of such continuations (rule unlock). After testing this rule, we observed that the generated code is not correct as the lock should only be acquired after the session is actually started.

Once the problem was identified, it was easy to correct the SmPL transformation rules by restricting the set of affected continuations, resulting in the rules shown previously.

Improving the transformations when using Coccinelle, as highlighted in the previous texts, comes down to extending a rule or writing an additional rule for dealing with a special case. In the original implementation of the MTL compiler, however, rewriting the generation of lock code requires correlating portions of code in several files. Among these files are those that are relevant to the implementation of the traversal of the language AST and the pretty printer of the generated code. The tasks of revisiting these files when bugs are found can become tedious, over the long term, even for those who implemented the compiler in the first place.

6. RELATED WORK

A number of surveys have considered the advantages and disadvantages of various DSL implementation strategies. Spinellis [27], Kosar [11], and Mernik [28] have categorized various DSL design and implementation patterns and have compared the different approaches in terms of implementation effort and end-user effort. For instance, Kosar *et al*. have ranked the *internal* (*embedded*) implementation approach as the best in terms of development effort, by showing that it requires less code. On the other hand, the *source-to-source* and *compiler generator* approaches require less effort from the end user, because the DSL syntax can typically be more closely tied to the domain, which often reduces DSL program sizes. In our work, we are able to provide a syntax that is tied to the domain, as network protocol developers are used to C programming, but that follows the internal implementation approach. We present in the succeeding texts other related work on DSL development.

DOI: 10.1002/spe

Internal domain-specific languages. Internal DSLs, also often referred to as domain-specific embedded languages are increasingly common, driven by the introduction of DSL-development methodologies based on general purpose languages (GPLs) such as Lisp, Ruby, Haskell, or Scala that are well adapted to serve as host languages [26, 29, 30]. Indeed, these GPLs provide a number of features that are beneficial in domain-specific embedded language implementation: support for higher-order functions, lazy evaluation, strong typing with polymorphism, and overloading [28]. Baars *et al*. have also proposed a number of embedded compilers for internal languages hosted in such languages [31–33], furthermore achieving run-time compilation of the internal language [34, 35]. However, their work requires explicit manipulation of the abstract syntax trees by the DSL developer, which can make debugging of the DSL implementation tedious.

Recent meta-programming techniques, such as expression templates, have facilitated the hosting of several DSLs in C++, including languages dedicated to scientific computing [10, 36]. Tratt has recently described an approach of DSL implementation through embedding into the Converge[§§] programming language [37]. In this case, the embedding is facilitated by a compile-time meta-programming feature. Our approach instead leverages a system, Coccinelle, that is external to the C host language. The Helvetia [38] workbench accommodates language embedding by providing an infrastructure that defines extension points for leveraging the host language's compiler and tools. Similarly, Hofer *et al*. have advocated polymorphic embedding [39] of DSLs to reconcile the rapid prototyping that can be achieved by simply borrowing the syntax and semantics of the host language in pure embedding with the flexibility that can be attained by using external toolchains. We also aim for this goal by leveraging the Coccinelle transformation engine as a tool chain for C-like programs.

The C language is a challenging target for DSL hosting. Existing approaches have typically relied on the use of macros and domain-specific libraries [40]. With this strategy, any check requiring control or data flow information must rely on externally developed analyses, requiring substantial development effort. We have avoided the need to develop such analyses from scratch by leveraging an existing scriptable code matching and transformation tool.

Compiler-based domain-specific languages. The design and implementation of compilers for DSLs is broadly discussed in the literature [6, 9, 27, 41]. Whether developers use standard compiler/interpreter techniques to directly implement a DSL or extend an existing GPL compiler, the development is often at a high cost and produces an implementation that is tied to the language or to a domain, complicating reuse. Despite these disadvantages, the compiler approach has gained wide acceptance because it allows constructing a DSL whose syntax is as close as possible to the notation used by domain experts and because it may offer good error reporting [11]. In our work, we have shown that a compiler toolkit based on Coccinelle is more flexible for DSL maintenance.

The Broadway compiler [42] allows developers to use code annotations to provide the compiler with domain-specific knowledge, rather than hard coding the knowledge in the compiler. The Broadway compiler, however, is more targeted towards optimizing uses of domain-specific libraries, rather than towards DSLs.

Program transformation. Approaches to implementing DSL compilers on the basis of syntactic rewriting or transformation rules have been proposed in the literature. Systems such as ASF+SDF [43], JTS [44], DMS [45], and Stratego [46–48] make it possible to design and implement DSLs on the basis of program transformation strategies. These systems often use specialized metalanguages to describe the various aspects of the DSL. The main advantage of our approach lies in the proximity of the language describing the transformation rules to the developed DSL and the developer effort required in learning the former. Stratego optionally allows developers to express patterns on concrete terms using concrete syntax, but the connections between these terms are expressed using a separate tree traversal language, creating a gap between the source program and the Stratego specification that transforms it.

[§§]http://convergepl.org/

DOI: 10.1002/spe

The Glasgow Haskell Compiler [49] is a source-to-source compiler that consists of specified transformation rules for performing inlining, dead code elimination and other simple optimizations. Glasgow Haskell Compiler, however, only addresses Haskell, rather than DSLs in which individual constructs may have a richer, domain-specific semantics.

Modular compilation. In the nanopass framework [50], the implementation of a compiler is structured into many passes, each performing very little work, to simplify development, testing, and debugging. This approach is very close to ours as it allows the implementation steps to reflect the organization of the analysis, transformations, and optimizations, facilitating understanding and maintenance. Our approach goes further by allowing the implementation of each compiler pass to be syntactically close to the generated code.

Lacey *et al*. pioneered the use of temporal logic to express and reason about compiler optimizations [51]. Inspired in part by this work, the implementation of Coccinelle uses temporal logic in the implementation of the '. . . ' operator. Thus, it is in a sense natural to use Coccinelle in a compiler implementation, providing a front end to the reasoning about control flow graphs embodied in temporal logic.

7. CONCLUSION AND FUTURE WORK

Developing a compiler for a DSL is complex, requiring both expertise in compiler construction techniques and domain knowledge. Most compilers are implemented in a monolithic way where the organization of domain-specific analysis and transformations is mingled, thus complicating maintenance. The structure of the compiler's code is furthermore often distant from the input and output programs, which can be a disadvantage during testing and debugging.

In this paper, we have demonstrated the suitability of specifying a DSL compiler using control-flow-sensitive concrete syntax-based matching rules. We embed domain-specific notations into the C programming language and rely on Coccinelle, a program matching and transformation tool, to carry out the required checks and transformations. In our source-to-source compiler approach, a DSL program is incrementally processed by various transformation rules to produce the final, safe, and optimized C program, which is compilable using a standard C compiler.

We have reported a successful experience of compiler development for a network protocol gateway DSL with the proposed approach, detailing its benefits over our previous compiler implementation. These benefits mainly involve the simplification of debugging and maintenance tasks, due to the use of transformation rules that are specific to the affected language constructs. This feature makes the rules independent of the set of constructs used in the rest of the code, potentially allowing some parts of a DSL program to be implemented as arbitrary C code. This makes it possible to explore the spectrum between the expressiveness of a complete C solution and the robustness of a complete DSL solution.

This work raises several potential research directions. Debugging is known to be difficult for languages implemented by translation, because there is no obvious connection between the executed code and the source code. One possibility is to introduce original source line information by using macros during the code reorganization phase, to improve error reporting. Another potential research direction is to improve the performance of Coccinelle, in the case of the application of a series of rules to a single code base, as required by our approach. Finally, we will investigate whether there are other DSLs that can benefit from a Coccinelle-based compiler.

REFERENCES

1. Kieburtz RB, McKinney L, Bell JM, Hook J, Kotov A, Lewis J, Oliva DP, Sheard T, Smith I, Walton L. A software engineering experiment in software component generation. *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, Berlin, Germany, 1996; 542–552.
2. Kosar T, Oliveira N, Mernik M, Pereira MJV, Črepinšek M, da Cruz D, Henriques PR. Comparing general-purpose and domain-specific languages: an empirical study. *Computer Science and Information Systems* May 2010; **7**(2):247–264.

DOI: 10.1002/spe

3. Kosar T, Mernik M, Carver JC. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering* June 2012; **17**(3):276–304.
4. Bromberg YD, Réveillère L, Lawall JL, Muller G. Automatic generation of network protocol gateways. *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware'09, Urbana Champaign, IL, USA, 2009; 1–20.
5. Bissyandé TF, Réveillère L, Bromberg YD, Lawall JL, Muller G. Bridging the gap between legacy services and web services. *Proceedings of the 11th ACM/IFIP/USENIX International Conference on Middleware*, Middleware'10, Bangalore, India, 2010; 273–292.
6. Fowler M. *Domain Specific Languages*, 1st edn. Addison-Wesley Professional: Upper Saddle River NJ, Boston, 2010.
7. Rhiger M. A foundation for embedded languages. *ACM Transactions on Programming Languages and Systems* May 2003; **25**:291–315.
8. Cunningham HC. A little language for surveys: constructing an internal DSL in Ruby. *Proceedings of the 46th Annual Southeast Regional Conference*, ACM-SE 46, Auburn, Alabama, USA, 2008; 282–287.
9. Leijen D, Meijer E. Domain specific embedded compilers. *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL)*, Austin, TX, USA, 1999; 109–122.
10. Prud'homme C. A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations. *Scientific Programming* April 2006; **14**:81–110.
11. Kosar T, Martínez López PE, Barrientos PA, Mernik M. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology* April 2008; **50**:390–405.
12. Aho AV, Sethi R, Ullman JD. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1986.
13. Réveillère L. Développement de systèmes distribués efficaces: Une approche fondée sur les langages métiers, November 2011. Hdr, Université Sciences et Technologies - Bordeaux I. Available at: http://hal.archives-ouvertes.fr/tel-00814406 [last accessed 26 August 2013].
14. Burgy L, Réveillère L, Lawall JL, Muller G. A language-based approach for improving the robustness of network application protocol implementations. *26th IEEE International Symposium on Reliable Distributed Systems*, Beijing, 2007; 149–158.
15. Mérillon F, Réveillère L, Consel C, Marlet R, Muller G. Devil: an IDL for hardware programming. *Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, USA, 2000; 17–30.
16. Muller G, Lawall JL, Duchesne H. A framework for simplifying the development of kernel schedulers: design and performance evaluation. *High Assurance Systems Engineering Conference (HASE)*, Heidelberg, Germany, 2005; 56–65.
17. Consel C, Hamdi H, Réveillère L, Singaravelu L, Yu H, Pu C. Spidle: a DSL approach to specifying streaming application. *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, Erfurt, Germany, 2003; 1–17.
18. Mernik M, Žumer V. Incremental programming language development. *Computer Languages Systems & Structures* April 2005; **31**(1):1–16.
19. Padioleau Y, Lawall JL, Hansen RR, Muller G. Documenting and automating collateral evolutions in Linux device drivers. *Proceedings of the 4th ACM European Conference on Computer Systems (EUROSYS)*, Glasgow, Scotland, 2008; 247–260.
20. Krishnamurthi S, Hopkins PW, McCarthy J, Graunke PT, Pettyjohn G, Felleisen M. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation* 2007; **20**(4):431–460.
21. Wand M. Continuation-based multiprocessing. *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, Stanford University, California, USA, 1980; 19–28.
22. Perrig A, Szewczyk R, Tygar JD, Wen V, Culler DE. Spins: security protocols for sensor networks. *Wireless Networks* September 2002; **8**(5):521–534.
23. Thekkath CA, Nguyen TD, Moy E, Lazowska ED. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking* October 1993; **1**(5):554–565.
24. Liu A, Ning P. Tinyecc: a configurable library for elliptic curve cryptography in wireless sensor networks. *International Conference on Information Processing in Sensor Networks*, IPSN '08, St. Louis, Missouri, USA, 2008; 245–256.
25. Hudak P. Modular domain specific languages and tools. *Proceedings of the 5th International Conference on Software Reuse (ICSR)*, Victoria, BC, Canada, 1998; 134–142.
26. Odersky M, Spoon L, Venners B. *Programming in Scala: A Comprehensive Step-by-Step Guide*, 1st edn. Artima Incorporation: USA, 2008.
27. Spinellis D. Notable design patterns for domain specific languages. *Journal of Systems and Software* February 2001; **56**(1):91–99.
28. Mernik M, Heering J, Sloane AM. When and how to develop domain-specific languages. *ACM Computing Surveys* December 2005; **37**:316–344.
29. Dubochet G. On embedding domain-specific languages with user-friendly syntax. *Proceedings of the 1st Workshop on Domain-Specific Program Development*, Nantes, France, 2006; 19–22.
30. Tate B, Hibbs C. *Ruby on Rails: Up and Running*. O'Reilly Media, Inc.: Sebastopol, California, USA, 2006.

DOI: 10.1002/spe

31. Baars AI, Swierstra SD. Typing dynamic typing. *Proceedings of the Seventh ACM Sigplan International Conference on Functional Programming*, ICFP '02, Pittsburgh, PA, USA, 2002; 157–166.
32. Baars AI, Swierstra SD. Type-safe, self inspecting code. *Proceedings of the 2004 ACM Sigplan Workshop on Haskell*, Haskell '04, Snowbird, Utah, USA, 2004; 69–79.
33. Baars AI, Löh A, Swierstra SD. Functional pearl: parsing permutation phrases (functional pearl). *Journal of Functional Programming* 2004; **14**(06):635–646.
34. Baars AI, Swierstra SD, Viera M. Typed transformations of typed abstract syntax. *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, Savannah, GA, USA, 2009; 15–26.
35. Baars A, Doaitse Swierstra S, Viera M. Typed transformations of typed grammars: the left corner transform. *Electronic Notes in Theoretical Computer Science* September 2010; **253**(7):51–64.
36. Di Pietro DA, Veneziani A. Expression templates implementation of continuous and discontinuous galerkin methods. *Computing and Visualization in Science* September 2009; **12**:421–436.
37. Tratt L. Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems* October 2008; **30**(6):31:1–31:40.
38. Renggli L, Gîrba T, Nierstrasz O. Embedding languages without breaking tools. *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, Maribor, Slovenia, 2010; 380–404.
39. Hofer C, Ostermann K, Rendel T, Moors A. Polymorphic embedding of DSLS. *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, Nashville, TN, USA, 2008; 137–148.
40. Hyde RL. The RATC v3.0 domain specific embedded language for C/C++ programmers.
41. Consel C, Latry F, Réveillère L, Cointe P. A generative programming approach to developing DSL compilers. *Fourth International Conference on Generative Programming and Component Engineering (GPCE)*, Tallinn, Estonia, 2005; 29–46.
42. Guyer S, Lin C. Broadway: a compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE* February 2005; **93**(2):342–357.
43. van den Brand MGJ, Deursen Av, Heering J, Jong HAd, Jonge Md, Kuipers T, Klint P, Moonen L, Olivier PA, Scheerder J, Vinju JJ, Visser E, Visser J. The ASF+SDF meta-environment: a component-based language development environment. *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, Genova, Italy, 2001; 365–370.
44. Batory D, Lofaso B, Smaragdakis Y. JTS: tools for implementing domain-specific languages. *Proceedings of the Fifth International Conference on Software Reuse*, Victoria, BC, Canada, June 1998; 143 –153.
45. Baxter ID, Pidgeon C, Mehlich M. DMS : program transformations for practical scalable software evolution. *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, Scotland, UK, 2004; 625–634.
46. Visser E. Stratego: A language for program transformation based on rewriting strategies. system description of stratego 0.5. In *Rewriting techniques and applications (RTA'01)*, Vol. 2051, Middeldorp A (ed.), Lecture Notes in Computer Science. Springer-Verlag: Utrecht, The Netherlands, 2001; 357–361.
47. Hemel Z, Kats LCL, Visser E. Code generation by model transformation. *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, ICMT '08, Zurich, Switzerland, 2008; 183–198.
48. Groenewegen DM, Hemel Z, Kats LC, Visser E. WebDSL: a domain-specific language for dynamic web applications. *Companion to the 23rd ACM Sigplan Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, Nashville, TN, USA, 2008; 779–780.
49. Peyton Jones SL, Santos ALM. A transformation-based optimiser for Haskell. *Science of Computer Programming* September 1998; **32**:3–47.
50. Sarkar D, Waddell O, Dybvig RK. A nanopass infrastructure for compiler education. *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, Snow Bird, UT, USA, September 2004; 201–212.
51. Lacey D, Jones ND, Van Wyk E, Frederiksen CC. Proving correctness of compiler optimizations by temporal logic. *POPL*, Portland, Oregon, USA, 2002; 283–294.

DOI: 10.1002/spe