Crossroads
The ACM Student Magazine

C-Transformers: A Framework to Write C Program Transformations

by ***Alexandre Borghi***, ***Valentin David***, and ***Akim Demaille***

Project Site: **http://transformers.lrde.epita.fr**
EPITA Research and Development Laboratory: **http://www.lrde.epita.fr**

Introduction

New trends in programming languages present a new challenge to researchers, productivity. One trend focuses on providing programmers with more productive languages. To this end, program transformation techniques are extremely powerful. To implement these transformations, language-specific frameworks are needed. To compose these frameworks, language agnostic tools are needed that can be used to quickly address a particular issue in any given language.

Program transformations encompass virtually every type of program processing. **Data extraction** is one form of processing that, for instance, calculates code metrics such as line and statement counts. Automatic document generation is also a form of data extraction. **Software renovation** pertains to improving existing code from possibly very large and ancient programs. Even with a thorough design, experienced programmers, and high level development tools, refactoring is often needed for healthy system development. Programmers typically perform refactoring by hand but today much of this process can be automated including the use of advanced constructs such as design patterns [**34**]. The development of refactoring tools for Integrated Development Environments (IDE) is an active area of research. **Optimization**, including domain-specific optimizations [**3**] and standard optimizations [**24**] is an important program transformation as well. **Translations**, such as compilation, typically make use of program transformation techniques with several refining steps. Generally, transformations are written in a standard language like C, C++, or Caml but employ only a few dedicated techniques as with Tiger in Stratego [**31**] or even the Stratego Compiler itself [**30**]. As a specific instance of translation, **language extension** provide existing languages with additional features and a translator that compiles, **assimilates**, [**5**]) them down to the original language. This paper will focus on such an application.

Language-specific frameworks.

Writing a compiler is a tremendous task and the implementation of a transformation framework is no exception. One needs to read and parser the input, possibly perform some disambiguation and type checking, perform the core transformations, and finally use a pretty printer to convert the program back to text. Every step of this framework is language-specific. The surrounding infrastructure can outweigh the transformation component by far. Therefore in order to make the

transformation implementation productive, language-specific frameworks are needed. In this paper we present the C-Transformers project [17], a framework enabling the seamless implementation of transformations programs in C and C variants.

Language generic components.

Vast research has been done and many tools developed to provide the technology needed to implement each type of component of the framework, sometimes leading to nice generalizations. For instance, there are tools that generate parsers from grammars. It is less known though that other steps also enjoy the existence of language generic components that can easily be tailored to a specific language [8]. C-Transformers uses Stratego/XT [13,32] for its library of language generic components. These off-the-shelf tools allow us to focus directly on specific C language issues such as disambiguation.

The C-Transformers is a free software project available on the Internet [17] developed by EPITA undergraduate students. EPITA is a French private engineering school dedicated to computer science. The Research and Development Laboratory of EPITA, LRDE, recruits amongst the best students willing to follow a more academic curriculum, possibly leading to a PhD. While members of the LRDE, they work on research topics supervised by assistant professors. V. David and A. Borghi are members of this group working actively on disambiguation under the supervision of Akim Demaille.

The Olena [10,18] and Vaucanson [16,20] projects gave the LRDE a strong experience in the development of C++ fast and generic libraries. New C++ programming techniques were then invented [6] which unfortunately resulted in somewhat obfuscated code. The Transformers project was created to address this issue, for instance by adding syntactic sugar to C++. C-Transformers can be used to implement virtually any kind of C program transformation but currently cannot be used for software renovation because of its lack of a reversible pre-processor.

Related Work

Program transformation has recently drawn a lot of attention these days. There are several existing projects. CIL [23,22] is an extremely complete and mature front end for the C language. It includes a type checker and a normalization of C that is moving towards a cleaner and simpler subset of the language. This considerably eases the transformation of C programs by reducing the number of cases to handle. Nevertheless CIL does not and cannot provide some of the features that prompted the development of C-Transformers. Transformations for CIL are more naturally expressed in Caml, the language it is written in. This does not prevent using CIL as a front-end to a Stratego program, but additional coding is required to do so. CIL's grammar is hardwired and thus cannot be easily extended, especially not in a modular way. Using an Syntax Definition Formalism (SDF) grammar and using SDF tools is necessary to enjoy the concrete syntax in Stratego. Furthermore CIL's output is not syntactically faithful; the resulting program is semantically equivalent but it is a deep modification of the input program, with a complete loss of layout, comments, and preprocessor directives. These characteristics make CIL inadequate for code factoring.

In the words of its authors [5,4], "MetaBorg provides generic technology for allowing a host language (collective) to incorporate and assimilate external domains (cultures) in order to strengthen itself. The ease of implementing embeddings makes resistance futile". Basically the MetaBorg project has exactly the same purpose as the Transformers project but on a different host language. Transformers focuses on C and C++ while MetaBorg started with Java. Java is a much cleaner language

than C, let alone C++ for which the development of a parser and disambiguation filter proved to be a daunting task. The MetaBorg chain relies on the similar tools except for semantics driven disambiguation. In [4], the power of the system is demonstrated by several extensions of Java to include Domain Specific Languages (DSL). Contrary to Transformers, they already have a type checker which allows syntax to be simplified even further by taking types into account during the disambiguation.

The goals of the Proteus [33] are extremely similar to Transformers; build a C transformation framework that makes it possible to preserve the programming style. These projects have many similar tools but differ on some key aspects. We strictly adhere to the standard grammar while they tailored theirs. As well, we have a workable solution to disambiguate C and extended C Abstract Syntax Trees (AST) but their paper does not mention disambiguation. Furthermore, we stick to Stratego with C concrete syntax while they introduce another language, YATL, compiled into Stratego. We explicitly want to experiment with grammar extensions while their work focuses on standard C program refactoring. Finally Transformers is free software [17]. The initial development effort also differs. Transformers attacks the modular disambiguation of C and C++ first; Proteus first makes sure that they can preserve the programming style, not only comments and layout but preprocessor directives too.

The Transformation Chain

In this section we present the C-Transformers framework, component by component. Figure 1 schematizes the whole process. Given a C grammar written in SDF, the SGLR parser reads the text and yields a set of parse trees, a parse forest. A disambiguation step keeps a single parse tree, transformed into an AST suitable for transformations. Finally the AST is converted back into compiler-ready C source text using pretty-printing.

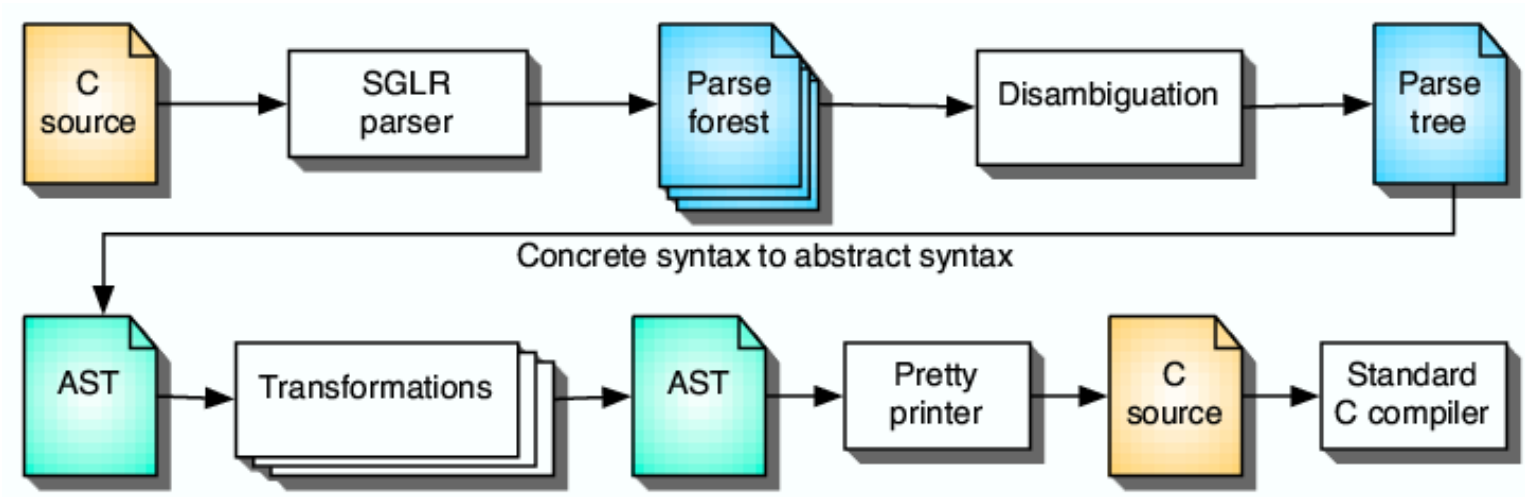

Fig. 1: The C-Transformers chain

SDF Grammars

The Stratego/XT tool set uses SDF [28] as its backbone. The tools are parametrized by an SDF grammar specifying the language at hand. In other words, grammars are contracts [8] thus they must be carefully crafted. The grammar syntax is modular which improves maintainability and extensibility by splitting the grammar in several modules. As well, additional information can be packed in the grammar via annotations.

Roughly the grammar can be written following two guide lines. It can be designed to be simple, making full use of SDF capabilities to handle precedence. This is an attractive approach since it results in rather short and elegant ASTs. We selected to stick rigorously to the grammar defined in the ISO C standard [**12**] in order to guarantee our strict compliance with the standard and to provide an environment of choice to experiment extensions to the standard which is precisely the theme covered in our **Case Study**. As a result our ASTs are somewhat more convoluted. For instance the AST for a simple `return 42;` is 26 nodes deep.

The C grammar counts 126 symbols and 356 rules. To ease the maintenance, the grammar is split into 53 small, manageable, sub-grammars. The boundaries of these sub-grammars were chosen to address coherent, atomic, related issues; they are finer than those of the standard which breaks the grammar in only 4 parts [**12**]. Figure **2** demonstrates some of SDF features.

```
module Declarators
imports ConstantExpressions TypeQualifiers ParameterDeclarations
exports
  sorts Declarator DirectDeclarator Pointer PointerSeq
  context-free syntax
    PointerSeq? DirectDeclarator              -> Declarator
    Identifier                                -> DirectDeclarator
    "(" Declarator ")"                        -> DirectDeclarator
    DirectDeclarator "[" TypeQualifierList? AssignmentExpression? "]"
                                              -> DirectDeclarator
    DirectDeclarator "[" "static" TypeQualifierList? AssignmentExpression "]"
                                              -> DirectDeclarator
    DirectDeclarator "[" TypeQualifierList "static" AssignmentExpression "]"
                                              -> DirectDeclarator
    DirectDeclarator "[" TypeQualifierList? "*" "]"
                                              -> DirectDeclarator
    DirectDeclarator "(" ParameterTypeList ")" -> DirectDeclarator
    DirectDeclarator "(" IdentifierList? ")"   -> DirectDeclarator
    Pointer+                                  -> PointerSeq
    "*" TypeQualifierList?                    -> Pointer
```

Fig. 2: SDF excerpt of the C grammar

Contrary to the (E)BNF, production rules are employed as reduction rules. This excerpt of a C grammar module focuses on function declarators such as signatures that are used both to declare and to define functions. The interface of the module specifies that it provides the symbols `Declarator` , ..., `PointerSeq`, and requires from other modules the symbols `ConstantExpressions`. The running example of **Case Study** will extend `Declarator` to support an additional form of function declaration.

SGLR and Parse Forests

A technology supporting ambiguity and yielding parse forests is needed. Amongst available techniques Generalized LR (GLR) is most attractive [**25**]. A generalized parser relieves us from obfuscating the grammar to cope with the limitations of the parsing technology such as the infamous shift/reduce or reduce/reduce conflicts. Even more importantly, a generalized parser is actually

indispensable to have the necessary level of modularity (see **Discussion**). Stratego/XT uses the state-of-the-art SGLR [**29**] parser. It provides all the required features, modularity and ambiguity support, and produces parse forests efficiently encoded using the ATerm library [**26**]. Actually, it is an ambiguous parse tree that is built, with `amb` nodes grouping alternative parses in a more useful way than genuine parse forests.

As another consequence from having chosen to use the ISO C99 standard grammar verbatim, we inherit its syntactic ambiguities, some of them being gratuitous, others requiring more powerful context sensitive disambiguation techniques. The most typical context sensitivity of C is its dependency on identifier types. For instance depending on the kind of entity the identifier a was associated with, `(a) * (b)` might cast `*b` to type a, or, multiply a by b. Symbols denoting unary and binary operators, such as `-`, `+` and `&`, exhibit the same ambiguity.

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

Fig. 3: Hello world, a famous first C program

Consider Figure 3 as a running example. The source code of this program is the input provided to SGLR, which will result in a parse forest. Besides the numerous ambiguities within `stdio.h` file, there is one in the `main` part. `printf`, according to the ISO C99 standard grammar, can be either the name of an enumeration constant or an identifier such as for a variable, type, or function. Figure **4** precisely shows the two production rules in competition.

Disambiguation

In the tradition of Yacc, context sensitive ambiguities are addressed by an elaborate cooperation between the parser and the scanner by maintaining a common table of symbols. This approach for example determines if the identifier a denotes a type or a variable. This results in a deterministic parsing where at most one parse tree is found. On the contrary, in the SGLR approach the ambiguous AST is traversed to gather context-sensitive information used to prune invalid parses, an approach called semantics driven disambiguation by [**27**].

The C-Transformers projects aims at modularity and extensibility. As such, we want to embed the disambiguation filters in the SDF grammar thus enjoying modularity for free. Another goal of this project is to relieve of the burden as possible from the programmer on specifying the order and types of tree traversals. Attribute Grammars (AG) fit very well these constraints.

Attribute grammars [**15**] are a formalism that supports syntax directed semantic analysis. Each grammatical rule is decorated with a set of equations that relate a node's attributes with those of its neighbors. AGs allow to focus on local aspects, leaving the global evaluation order aside, under the responsibility of a generic engine. Although AGs cannot modify the trees, their use for disambiguation is straightforward. Attributes convey information by using for example a symbol table. Conflicting

branches of the parse forest are flagged, and a language generic filter is run afterward on the parse forest, pruning inconsistent alternatives. Since no AG engine existed for SDF, we developed one. Attribute rules are embedded in the SDF grammar as additional annotations.

```
Identifier -> PrimaryExpression
       {attributes(disamb:
          root.ok := <lookup> (Identifier.string, root.lr_table_in)
                     ;  (Variable <+ Function)
       )}

Identifier -> EnumerationConstant
       {attributes(disamb:
          root.ok := <lookup> (Identifier.string, root.lr_table_in)
                     ;  Enum
       )}

Identifier -> TypedefName
       {attributes(disamb:
          root.ok := <lookup> (Identifier.string, root.lr_table_in)
                     ;  Typedef
       )}
```

Fig. 4: AG-driven disambiguation

Figure **4** demonstrates the use of AG to disambiguate C. This example focuses on an ambiguity of C. For example, an identifier `foo` might denote a variable, a function, a value of an enumeration type, or a type name. When traversing a node of this type, it is important to ensure that the `Identifier` was declared to be a variable or a function. If not, the node must be marked a `root` attribute `ok` to be failed. This will prune an incorrect alternative from the associated ambiguity node. The other cases are similar. Note that the table `lr_table_in` is automatically propagated from left to right. The performance of the system is very satisfying. Disambiguation is negligible compared to the whole parsing, including conversion into AST.

In Figure **3**, each ambiguity branch of the ambiguity will be evaluated. The interpretation of `printf` is one such ambiguity according to the rules of Figure **4**. During the traversal of the `stdio.h` part, `printf` will be recorded in `lr_table_in` as declaring a `Function`. During the traversal of the `main` part, the disambiguation rules of Figure **4** will therefore flag the valid derivation `Identifier -> PrimaryExpression`. A small auxiliary tool will prune all the invalid options afterward, yielding a unique parse tree. This parse tree which includes all the details about the layout, comments, exact characters that were used is then simplified into a much more compact AST free from the layout, and lexical details.

Transformations

The C-Transformers project and its peer C++-Transformers are somewhat ill-named, since it does not directly address transformation. Rather it is a framework for implementing C program transformations. Any transformation system is suitable, provided it supports our format for ASTs. Amongst the possible engines to express transformations, we particularly like the Stratego

programming language [1]. In the Figure 3 example, a transformation could be performed on the AST. For example, replacing the call to `printf` by a call to the faster `fputs` function.

In Stratego every piece of data is a **term**, an abstract syntax tree. **Conditional rewriting rules** specify how a particular tree matching a specific shape should be transformed. A specific transformation, such as translating extended C to C, involves several rewriting rules to apply at different places of the tree, and in a specific order. **Rewriting strategies** provide an elegant means to control when and where to apply rewriting rules. In addition, a rich set of operators allows to build arbitrarily complex transformations (i.e., strategies) from simple atomic ones.

Many transformations are sensitive to the static scoping rules of the target language. For instance α-conversion, the renaming of variables, must assign different names to the same identifier occurring in different scopes. **Dynamic rewriting rules** handle scopes gracefully. They can be created at any time but have their existence bound by scopes. To perform α-conversion, traverse from left to right, and for each variable declaration create a rewriting rule that maps the identifier to a fresh one. Entries and exits of scopes trigger the creation and destruction of the associated dynamic rules. This is much simpler than having to write generic static rules dealing with tables of symbols.

Thanks to a tight integration with SDF, Stratego provides **concrete syntax** support. In a Stratego program, instead of having to refer to AST constructs (e.g., `MUL (ADD (INT (1), INT (2)), INT (3))`) one can directly use the (concrete) syntax of the target language (e.g., `(1 + 2) * 3`). Examples of Stratego are given the following case study.

Pretty-Printing

Pretty-printing is a conversion from abstract syntax to concrete syntax. In this work, it is performed by Generic Pretty-Printer (GPP) [9], driven by language specific tables. These tables are generated from the SDF grammar, with embedded handcrafted directives to improve the result. In the Figure 3 example, the final pretty-printed result of our chain would be very different from its input since instead of `#include <stdio.h>` one would have the whole content of the file.

Case Study

C-Transformers provides a simple and powerful framework to transform C programs. To demonstrate its capabilities, we extend C into ContractC: C with design by contract support. The C-Transformers framework will be used to compile ContractC down to ISO C.

Design by Contract

Design by Contract is a software design and implementation methodology invented and promoted by Bertrand Meyer for his language, Eiffel [21]. The starting point is to consider that a function call involves two parties, the supplier and the client. The signature of the function is a (weak) form of a contract:

- the types of the incoming argument(s) are requirements put on the client, the caller, by the supplier, the callee
- the type of the outgoing result(s) are guarantees given to the client by the supplier.

A successful function call requires that both parties respect their part of the contract. Statically typed programming languages enable statical checks, i.e., at compile time, while dynamically

typed languages delay the verification until execution time. Note that in addition the signature of the function is a weak form of documentation. For instance, the signature of the square-root function, `double -> double`, specifies that it requires and returns a floating point number. Such information is always provided either in the documentation or in comments in dynamically typed languages.

Design by Contract extends signatures to include predicates between incoming and outgoing arguments. For instance the square-root function requires a non negative argument, a precondition, and ensures that the square of its result equals the incoming argument, a postcondition. Support for and use of pre/postconditions dramatically improve the safety of programs, in particular when reusing components [**11**]. Take for an example, Eiffel promoters claim that design by contract could have avoided the failure of the Ariane 501 launcher [**14**]. To demonstrate the use of C-Transformers, in the following we add support for pre and postconditions to the ISO standard of C, based on the proposal of [**7**] for a C++ standard extension. The resulting language is here named ContractC.

Syntax

The adaption of the C++ extension proposal [**7**] to C results in adding support for pre and postconditions to function declarations, not implementation. Indeed, since pre and postconditions are pieces of formal documentation, they belong to the header file, which corresponds to the interface of a module in C parlance. Nevertheless, when compiled the contract is to be integrated in the implementation of the function. It is the called function which will ensure that pre and postconditions are properly met.

See Figure **5** for an example of ContractC, a set of pre and postconditions put on the function `sqrt`.

```
double sqrt (double r)
    precondition
    {
        r >= 0.;
    }
    postcondition (result)
    {
        result >= 0.;
        equal_within_precision (result * result, r);
    };
```

Fig. 5: An function declaration example in ContractC

To implement ContractC in **Transformers**, the first step is to extend its grammar with an additional rule for function declarations as shown in Figure **6**.

```
module PrePostConditions

imports Declarators

exports
  sorts
    DirectDeclarator ReturnValueDeclaration
    Assertion PostCondition PreCondition
```

```
  context-free syntax
    DirectDeclarator "(" ParameterTypeList ")"
      PreCondition? PostCondition?  -> DirectDeclarator
```

Fig. 6: Extension of the C grammar to support pre and postconditions. This rule is based on [7]

Compilation to C

```
double sqrt (double r)
{
  return _libc_sqrt (r);
}
```

Fig. 7: A C function implementation

```
double sqrt (double r) { double result; { assert(r >= 0.); } { result = _libc_sqrt
(r)); goto end; } end: { assert (result >= 0.); assert equal_within_precision (result
* result, r); } return result; }
```

Fig. 8: A C function implementation with contracts installed. This is the final result, when the contract specified in the function declaration from Figure 5 is inserted in the body of the function implementation in Figure 7

The ContractC compiler (towards C) translates the contract into code run by the supplier, the called function. As an example, the ContractC declaration of Figure 5 transforms the regular C implementation of Figure 7 into that Figure 8. Writing such a transformation in Stratego is simple: see Figure 9, Figure 10, and Figure 11.

```
prepost = io-wrap(alltd(FunDecl <+ Contract))
```

Fig. 9: The top-level of the transformation. In a single top-down traversal (`alltd`), for each function declaration with contract create a rule to install the contract, or for each function definition, install the contract

```
FunDecl:
  Decl|[ ret fn (args) pre post ; ]| ->
  Decl|[ ret fn (args) ; ]|
    where
      rules (Contract :
        FunDef|[ ret' fn (args') cpstm ]| ->
        FunDef|[ ret' fn (args') cpstm' ]|
          where <transform(|ret', pre, post)> cpstm => cpstm')
```

Fig. 10: Handling a ContractC function declaration. Each function declaration with a contract must be rewritten without, and create an additional instance of `InstallContract` dedicated to the current function name `fn`

```
InstallContract(|as1, as2, type, res):
  CmpdStm|[ body ]| ->
```

```
CmpdStm|[
   {
     type res;
     {
       as1
     }
     body'
   end:
     {
       as2
     }
     return res;
  }
]|
  where <alltd(ReturnToGoto(|res))> body => body'
```

Fig. 11: Installing the contract in the function implementation

Installing the contract takes more code to take several details into account; pass conditions to `assert`, gather the return type to declare the `result` variable, handle possible name clashes with its name, and replace `return` with assignment-and-goto. Ultimately, once converted the pre/postconditions assertions `as1`/`as2`, the `return` in the body, and the name of the result variable `res`, the function declaration is transformed.

Discussion

To be effective, the C-Transformers tool chain must be easily extensible; every component must be easily configurable. In other words, featuring modularity is not merely satisfying with regards to current programming mottos, it is a must-have for every single tool involved in the chain.

Contrary to the full class of context-free grammars, usual proper subclasses such as LL and LR are not stable under union. Therefore parsing technologies limited to LL or LR parsing do not meet our requirements of **modularity**. The scanner-less generalized LR parser, SGLR supports the full context-free class of grammars — and actually some more. In addition SDF provides powerful and convenient means to compose modules. The built-in support for (possibly ambiguous) AST construction also enables to focus on actual issues, instead of having to code lengthy AST support classes or to fight parser conflicts.

The disambiguation requires modularity in its strongest sense. For simple disjunctive union, disambiguation tools will not suffice. For instance, the sample ambiguities discussed in the introduction, `(a) * (b)`, obviously affect the pre and postconditions. Not only do we need to be able to add new disambiguation rules to those of the host language, but we also need to intermix them just as freely as for SGLR modules. Attribute grammars handle this gracefully, being modular by nature, and better yet, sharing the exact same definition of modularity as the grammar itself.

The transformation needs to focus on specific spots; it is not concerned by most of the nodes. Stratego provides generic traversal operators that not only relieve the programmer from tedious work but also guarantee the independence of the transformation from changes in the host grammar. In other words, Stratego also meets the modularity requirements.

Finally the pretty printing engine needs to support plug-ins to express the visual structure of the additional constructs. While GPP does support such add-ons, we find embedded pretty-printing rules more convenient. Being bound to the grammar, they share its modularity in the exact same sense, like attribute rules do.

If any of these components were to lack modularity support, or even slightly deviates from the standard set by SDF and SGLR, then it might not be possible to perform the transformation or it would require massive infrastructure. The result would also become extremely fragile: any change in the host grammar or in the extension can possibly require a full overhaul. C-Transformers features the full concept of modularity allowing concise and robust implementations of transformations.

Type	**Length**	**Comment**
Grammar	6 rules	Pre/postconditions, function declaration
Disambiguation	6 lines	Same as for regular function declaration
Pretty Printing	4 rules	Pre/postconditions, function declaration
Meta-variables	3 rules	Pre/postconditions, assertions
Transformation	60 lines	20 of which obey C formatting rules

Fig. 12: ContractC implementation effort

Although still in the early stages of development, C-Transformer is already very **usable**: Thanks to modularity the implementation of ContractC is quite straightforward and very compact. The volume of the full project is given in Figure **12**.

The whole processing chain is composed of several steps, each one adding its own overhead to the actual transformations. We first do preprocessing with an inhouse preprocessor equivalent to POSIX cpp but produces slices of the input. Parsing is then carried out by executing the SGLR on each slice. Concatenation is performed by pasting all the different slices together. Evaluation of the attributes are computed on the parse tree. Next pruning is conducted by removing the alternative ambiguities flagged as incorrect by the evaluation. A checking step makes sure no ambiguities remain. Implosion converts the transformations into a suitable form. This step builds an AST from the parse tree. The transformation compiles ContractC to C. Finally pretty printing translates the AST back to a text in concrete syntax.

	Queens		**Hello, World**		**Lemon**	
Lines of code	76		448		3550	
Duration	s	%	s	%	s	%
Preprocessing	0.11	15	0.13	1	0.18	0
Parsing	0.09	12	2.41	25	8.11	4
Concatenation	0.01	1	1.3	13	2.73	1
Evaluation	0.12	16	1.14	12	14.16	6
Pruning	0.08	11	0.8	8	8.15	4
Checking	0.01	1	0.2	2	1.53	1

Implosion	0.13	18	3.15	33	179.5	81
Transformation	0	0	0.02	0	0.35	0
Pretty-printing	0.18	25	0.53	5	6.95	3
Total	0.73	100	9.68	100	211.66	100

Fig. 13: Running time of the C-Transformers chain

Three examples were chosen to measure the contributions of each step. Queens is extremely short and includes no header. Hello World is presented in Figure **3**. Lemon is a parser generator that fits in a single C file. The great variation of the figures is due to the fact that these samples are quite small, the last one being the only significant example. Then the whole processing is dominated by the conversion of the parse tree in an AST. We conclude that the cost of our technology AG-driven disambiguation is negligible, and there is not even reasons to try to optimize it further. Unfortunately we also conclude that currently, parsing ambiguously grammars as ambiguous as those of C and C++ and then disambiguating is somewhat prohibitive.

We have been told that future versions of SGLR might perform the implosion during the parsing. Maybe some of the cost will be lowered but unless SGLR is also able to run AG-driven disambiguation, it will still have to build massive ASTs that will have to be pruned afterward.

Conclusion

We have presented modern program transformation techniques using C-Transformers as an example. To demonstrate the intrinsic power of these techniques, we implemented an extension to ISO C in an extremely reduced number of lines without sacrificing readability. We have emphasized how essential modular and language generic tools are.

In the future, several issues deserve more work. The current AG system works perfectly well with very satisfying performance, but many improvements are expected. The most notable addition will be additional syntactic sugar to cover the most common idioms we find during the implementation of semantics driven disambiguation filters. Currently the attribute rules are lengthy and repetitive. Up to date AG engines such as Utrecht University Attribute Grammar System (UU-AG) [**2**] abstract attributes from production rules which allows shorter and more readable declarations. Our AGs are written in SDF and transformed with Stratego/XT, the methodology described in this paper can be applied to them.

The most severe issue with C-Transformers is that it disrupts the programming style [**33**]. It produces preprocessed C with all the `#include` and other `#define` expanded. For instance, the simple "hello world" 4 line program (see Figure **3**) actually contains 450 lines of code coming from `#include <stdio.h>`. Not only does this clutter the result, it also makes it non portable. Indeed, much of the portability of C is handled by system headers that make wide use of operating system and architecture specific code. This issue can be prevent C-Transformers from being used in some possible applications. For instance when users wish to ship the product but not the producer, or when the result is meant to be maintained by humans. This limitation is temporary though. Our limited resources were assigned to address the disambiguation first, and then programming style will be preservable. To cope with this issue we planned to develop a reversible preprocessor that embeds annotations in the AST to allow their reversal, very much in the spirit of Proteus project in fact [**33**].

Once we have completely overcome the issues surrounding C, we will focus again on C++. C++ inherits ambiguities from C such as `(a) * (b)`, but it also adds ambiguities of its own, even when identifier kinds are known. For instance, let `T` be a type, depending on the context `T(a);` denotes either the declaration of the variable `a`, or a call to the constructor `T::T(a)`. The `template` mechanism makes the process complex and nauseam requiring not only to carry symbols in tables but also arbitrarily long ASTs. Our C++ grammar is complete and the disambiguation filter is almost completed and in the foreseeable future will also make use of the reversible preprocessor.

Acknowledgments

The Transformers was started by Robert Anisko while an LRDE student. Today it is maintained by other LRDE students under the supervision of Akim Demaille. Valentin David was the architect of the current disambiguation chain now maintained by Alexandre Borghi. Other LRDE students that have contributed significant portions of the Transformers project are Clément Vasseur, Nicolas Pouillard, and Olivier Gournet. The authors thank particularly Olivier Gournet who, behind the scenes, made ContractC work.

References

1

Stratego Language Home Page. http://www.stratego-language.org.

2

Arthur Baars, Doaitse Swierstra, and Andres Löh. UU-AG System, 1999. http://www.cs.uu.nl/wiki/Center/AttributeGrammarSystem.

3

Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 65--74, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

4

Martin Bravenboer, René de Groot, and Eelco Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, Braga, Portugal, July 2005.

5

Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365--383, Vancouver, Canada, October 2004. ACM Press.

6

Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss. A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL)*, Anaheim, CA, USA, October 2003.

7

Lawrence Crowl and Thorsten Ottosen. Proposal to add contract programming to C++ (revision 3). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1866.html, August 2005.

8

JongeMerijn de Jonge and Joost Visser. Grammars as contracts. In Greg Butler and Stan Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposion, GCSE 2000*, volume 2177 of *Lecture Notes in Computer Science*, pages 85--99, Erfurt, Germany, October 2001. Springer.

9

Merijn de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.

10

Alexandre Duret-Lutz. Olena: a component-based platform for image processing, mixing generic, generative and OO programming. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—-Young Researchers Workshop; published in "Net.ObjectDays2000"*, pages 653--659, Erfurt, Germany, October 2000.

11

ISE Software. Building bug-free O-O software: An introduction to design by contract(TM). http://archive.eiffel.com/doc/manuals/technology/contract/page.html, 1993.

12

ISO/IEC. ISO/IEC 9899:1999 (E). *Programming languages - C*, 1999.

13

Merijn Jongede Jonge, Eelco Visser, and Joost Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Perigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2001.

14

Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129--130, 1997. http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html.

15

Donald E. Knuth. Semantics of context-free languages. *Journal of Mathematical System Theory*, pages 127--145, 1968.

16

Sylvain Lombardy, Raphaël Poss, Yann Régis-Gianas, and Jacques Sakarovitch. Introducing Vaucanson. In Springer-Verlag, editor, *Proceedings of Implementation and Application of Automata, 8th International Conference (CIAA)*, volume 2759 of *Lecture Notes in Computer Science Series*, pages 96--107, Santa Barbara, CA, USA, July 2003.

17

LRDE — EPITA Research and Developpement Laboratory. Transformers home page, 2005. http://transformers.lrde.epita.fr.

18

LRDE. Olena home page, 1999. http://olena.lrde.epita.fr/.

19

LRDE. Tiger project home page, 2000. http://tiger.lrde.epita.fr/.

20

LRDE. Vaucanson home page, 2001. http://vaucanson.lrde.epita.fr/.

21

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, March 2000.

22

George C. Necula. CIL home page, 2005. http://sourceforge.net/projects/cil.

23

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213--228, London, UK, 2002. Springer-Verlag.

24 Karina Olmos and Eelco Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, volume 70 of *Electronic Notes in Theoretical Computer Science*, page 20, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.

25 Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

26 BrandMark van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259--291, 2000.

27 BrandMark van den Brand, Steven Klusener, Leon Moonen, and Jurgen J. Vinju. Generalized parsing and term rewriting: Semantics driven disambiguation. volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.

28 Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, July 1997.

29 Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

30 Eelco Visser. A bootstrapped compiler for strategies (extended abstract). In B. Gramlich, H. Kirchner, and F. Pfenning, editors, *Strategies in Automated Deduction (STRATEGIES'99)*, pages 73--83, Trento, Italy, July 5 1999.

31 Eelco Visser. Tiger in Stratego, 2002. http://www.program-transformation.org/Tiger/WebHome.

32 Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216--238. Spinger-Verlag, June 2004.

33 D. G. Waddington and B. Yao. High fidelity C++ code transformation. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, Electronic Notes in Theoretical Computer Science, Edinburgh University, UK, April 3 2005.

34 Mikal Ziane. Towards tool support for design patterns using program transformations. In *Langages et Modèles à Objets (LMO)*, volume 7, pages 199--214, Le Croisic, January 2001. Hermès Science Publications.

Biography

Alexandre Borghi (**Alexandre.Borghi@lrde.epita.fr**) is an LRDE **(http://www.lrde.epita.fr)** student.
He works on AG driven disambiguation as the current maintainer of the Transformers disambiguation chain.

Valentin David (**Valentin.David@lrde.epita.fr**) is a former LRDE **(http://www.lrde.epita.fr)** student working specially on C++ parsing in the Transformers project. He is now a PhD student at the University of Bergen, working on C++ transformation.

Akim Demaille (**Akim.Demaille@lrde.epita.fr**) is the director of the EPITA Research and Development Laboratory (**http://www.lrde.epita.fr**). He teaches language theory, compiler construction, and formal logic at EPITA. His research topics include program transformation with the Transformers project, automata theory with the Vaucanson project [**20**], and compiler construction pedagogy with the Tiger project [**19**].