Automating Proofs of C-- Programs

Carine Fédèle, Emmanuel Kounalis and Olivier Ponsini

I3S
Les Algorithmes
2000, route des Colles 06410 Biot, France

Abstract. This paper describes a method for deriving equations from programs written in C--, a simple imperative language. Programs are automatically transformed from the source code, without any user annotation, in a set of conditional equations, equivalent to the program and suitable to perform proofs in theorem provers.

1 Introduction

In most cases where a program specification is done correctly, software deficiencies that come from the gap between the specification and its actual coding are by far more numerous than errors due, for instance, to hardware failure or to the compiler. In order to increase confidence in code production, efforts should center on verifying that programs meet their requirements, that is, that they are sound with regard to their specification. When dealing with the problem of *program soundness*, developers usually tend to use empirical methods like test sets. But this is not sufficient for applications that need a high degree of reliability. This kind of applications strongly benefits from the use of *formal methods* for validation.

Formal methods are mathematical tools and techniques aimed at specifying and verifying software or hardware systems. By verification, we mean the analysis of a system so as to demonstrate it owns the desired properties. In this paper, we endeavor to work on source code produced by programmers, in contrast with systems that generate code from specifications. This way, the code can be manually optimized. To carry out this task, most proof systems ask the user to annotate the source code, mixing the specification content with the code. Therefore, either the programmer must have a good understanding of the specification language or the specifier sufficient knowledge in the coding language. In both cases, a same person must master two disparate languages and adopt two different points of view. To avoid this, we address the problem of a verification method which distinguishes the two activities — specifying and coding.

1.1 A Proof Framework

In a previous work [3], Fédèle and Kounalis introduced a framework for proving automatically properties of C-- programs, a small imperative language. The

idea was to translate source code into first order equations. These equations would constitute the axioms of a logic in which desired properties of the program, its specification, could be proved. Since both the axioms and the specification of the program would be written in equational logic, proofs could be conducted, whether automatically or no, through proof systems like *theorem provers*.

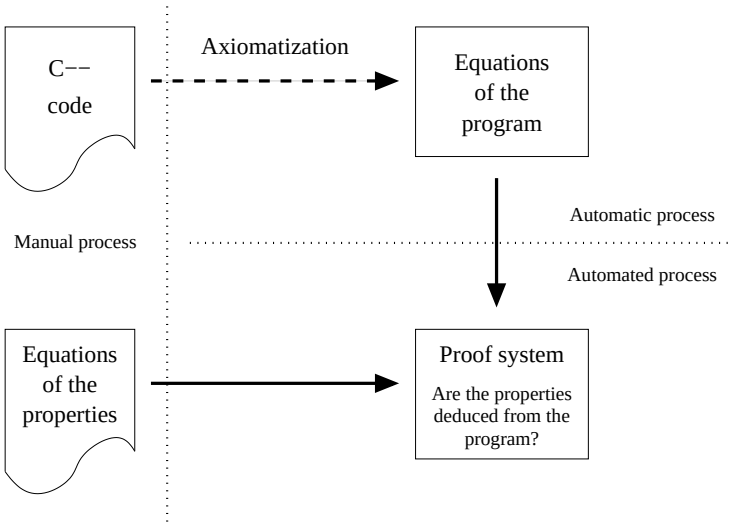

Fig. 1. Proof process overview

The proof process is shown in Fig. 1. Users write the C-- code of a program. They also write the program specification as a set of properties expressed in equational logic. The source code is then transformed automatically into equations. The equations of the program are seen as the axioms, and the properties as the theorems of a logic. To prove the theorems deduce themselves from the axioms is equivalent to prove that the program is correct regarding its specification. This last part is done automatically using *theorem provers* able to do mathematical induction like NICE [6] or interactively using *proof checkers* like COQ [2].

In this paper, we focus on the program axiomatization: the operation which derives equations from source code.

1.2 The C-- Language

For our experiments we use a very simple imperative language. The C-- syntax is similar to the C one. The main features of the language are:

- assignment;
- control flow statements: *if* ... *else*, *while* and *return*;

Listing 1. Identity function.

```
int identity (int x) {
  return x;
}
```

- two predefined types: integers (*int*), and lists of integer (*list*);
- usual arithmetic operators;
- operators on lists: *getHead* which returns the first element of a list, *getQueue* which returns a copy of a list except for the first element, *NULL* which represents an empty list, and *cons* which inserts an element at the beginning of a list.

Several common features to imperative languages are unavailable:

- no user's defined types allowed;
- no global variables allowed;
- no pointers directly accessible to the user — of course some are used in the predefined type list.

2 Axiomatization

The axiomatization is the operation which takes as input *a C-- program* and gives as output *a set of equations* semantically identical to the program. This transformation is done in three steps and without any user interaction. Figure 2 shows the steps involved in the axiomatization. The method is based on a *rewriting system* whose role is to semantically analyze programs. A rewriting system substitutes equal terms depending on a set of *rewriting rules* (see [7] for an introduction).

The goal of the first step is to provide a correct input for the rewriting, that is a *term* over the rewriting system signature — a signature is a set of function symbols and arities. This term is then rewritten into another term which is the *intermediate environment*. Finally, the *conditional equations* are extracted from the environment.

As an introductory example, let us see the different stages of the axiomatization of the function of listing 1. The *identity* function will be transformed in the term

$$\mathrm{GE}\left(\begin{array}{l}\mathrm{Cons\_inst}(\mathrm{Return}(\mathit{identity}, x), \mathit{End\_inst}),\\ \mathrm{Cons\_env}(\mathrm{Pair}(x, \mathrm{EP}(x)), \mathit{Empty\_env})\end{array}\right),$$

then, after rewriting, in the environment

$$\mathrm{Cons\_env}(\mathrm{Pair}(x, \mathrm{EP}(x)), \mathrm{Cons\_env}(\mathrm{Pair}(\mathit{identity}, \mathrm{EP}(x)), \mathit{Empty\_env})),$$

and, finally, in the equation

$$\mathit{identity}(x) = x \ .$$

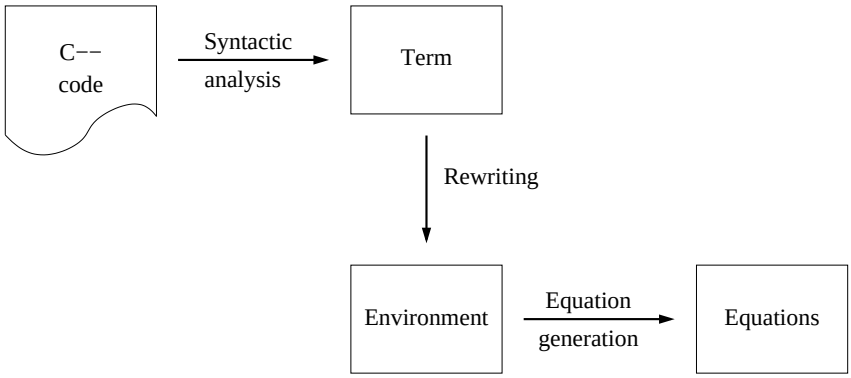

Fig. 2. Axiomatization process overview

2.1 Term Generation

`Input` : A C-- program.
`Output` : A term over the rewriting system signature.

A syntactic analysis is performed on each function of the C-- program to build a distinct term appropriate for rewriting.

- A C-- *function* is seen as a list of statements and an initial environment, contained in a GE[1] term. The initial environment is made up of the function formal parameters combine with EP[2] terms denoting the effective parameters — formal parameters behave like local variables to which are assigned the effective parameters.
- A *sequence* of statements is simulated by a list of terms — whose constructors are *Cons_inst* and *End_inst*.
- *Return* statements produce a pair linking the name of the function and its return expression.
- *Variable declarations* are considered as assignments added to the list of statements of the function. If a variable is not initialized, a default value is attributed.
- *Expressions* — or right-values — are not parsed.
- The other constructs of the C-- language are simply matched with equivalent terms of the rewriting system, as for instance:

$$\begin{aligned} x = y &\mapsto \mathrm{Assign}(x,\ y), \\ \mathrm{if}(c)\ \mathit{insts1}\ \mathrm{else}\ \mathit{insts2} &\mapsto \mathrm{If}(c,\ \mathit{inst\_list1},\ \mathit{inst\_list2}), \\ \mathrm{while}(c)\mathit{insts} &\mapsto \mathrm{While}(\mathit{while\_number},\ c,\ \mathit{inst\_list})\ . \end{aligned}$$

In the case of the *identity* function of listing 1, the GE term is composed of the single instruction $\mathrm{Return}(\mathit{identity}, x)$ and the initial environment $\mathrm{Pair}(x, \mathrm{EP}(x))$, since x is the unique function parameter.

[1] Generate Environment.
[2] Effective Parameter.

2.2 Environment Generation

`Input` : A term.
`Output` : An environment.

In this second step, the term produced by the code analysis is normalized[3] according to the rewriting rules. The resulting term is a C-- function environment.

Description. The rules of the rewriting system express the operational semantics of the C-- language. Thus, the environment produced by rewriting represents the distinct execution paths of the function, along with their associated conditions and the final expression of the variables and function. By execution paths we mean the ways in which statements can be executed in a function accordingly to control flow statements. The paths are enclosed in *Choice* terms. A *Branch* term associates a condition to a variables state. The state of the variables is represented by a list of terms Pair(*variable*, *value*). The complete definition of an environment is given by the grammar in Fig. 3.

For instance, the function of listing 2 defines two execution paths, one for the *if part* and one for the *else part*. Therefore, the environment will contain the term

$$\text{Choice}(\text{Branch}(\mathit{cond}, \mathit{if\_part}), \text{Branch}(\text{Not}(\mathit{cond}), \mathit{else\_part})) \ .$$

$$
\begin{array}{ll}
< \text{env} > & ::= < \text{choice} > \mid < \text{env\_elt\_list} >; \\
< \text{env\_elt\_list} > & ::= \mathit{Cons\_env}(< \text{env\_elt} >, < \text{env\_elt\_list} >) \mid \mathit{Empty\_list}; \\
< \text{env\_elt} > & ::= < \text{while\_closure} > \mid < \text{pair} >; \\
< \text{choice} > & ::= (< \text{branch} >, < \text{branch} >); \\
< \text{branch} > & ::= \mathit{Branch}(< \text{cond} >, < \text{env} >); \\
< \text{while\_closure} > & ::= \mathit{WC}(< \text{int} >, < \text{cond} >, < \text{env} >, < \text{var\_list} >); \\
< \text{pair} > & ::= \mathit{Pair}(< \text{var} >, < \text{value} >); \\
< \text{value} > & ::= < \text{exp} > \mid < \text{loop} >; \\
< \text{var\_list} > & ::= \mathit{Cons\_var}(< \text{var} >, < \text{var\_list} >) \mid \mathit{End\_var}; \\
< \text{loop} > & ::= \mathit{Loop}(< \text{int} >, < \text{var} >, < \text{exp\_list} >); \\
< \text{exp\_list} > & ::= \mathit{Cons\_exp}(< \text{exp} >, < \text{exp\_list} >) \mid \mathit{End\_exp}; \\
< \text{exp} > & ::= \text{an expression}; \\
< \text{cond} > & ::= \text{a condition}; \\
< \text{var} > & ::= \text{a variable}; \\
< \text{int} > & ::= \text{an integer};
\end{array}
$$

Fig. 3. Environment grammar.

[3] The term is rewritten until no more rule can be applied.

Listing 2. An alternative.

```
int alternative () {
if (cond)
  if_part
else
  else_part
 ...
}
```

While Statements. The semantics of *While statements* is quite specific. Indeed, each loop is considered as a separate recursive function with its own parameters and body. But a function can only return a single value, and yet several variables can be modified by a loop. So, for each variable which is modified in the loop body, a new loop function is defined and its return value is the modified variable one. In addition, since any variable of the function is likely to be used in the loop body, the loop function takes all the variables as parameters. Consequently, when a loop is encountered, the environment is modified as follows:

- A new WC[4] term containing all the information needed to generate the loop functions is created. The information is the number of the loop, the exit condition, the statements of the loop body and the list of all the variables.
- Each variable which is modified in the loop body is assigned a call to the corresponding loop function with the current state of the variables passed as parameter.

The following example shows how while statements are handled. Let us suppose a C-- function declares three variables — x, y, z — two of which are modified in a loop body, like in listing 3. During rewriting of the term corresponding to function f, the term

$$\mathrm{WC}(1, y > 0, \mathrm{GE}(\textit{the instruction list},\ \textit{the loop initial environment}), (x, y, z))$$

is created. The instruction list is composed of the two assignments modifying x and y. The loop initial environment is the list of pairs $(x, \mathrm{EP}(x))$, $(y, \mathrm{EP}(y))$ and $(z, \mathrm{EP}(z))$. The term GE means that a new environment will be evaluated for the loop body. This will lead to the definition of two functions: LOOP^1_x and LOOP^1_y.

$$\begin{cases} \text{If } y > 0 \text{ then } \mathrm{LOOP}^1_x(x, y, z) = \mathrm{LOOP}^1_x(x + z, y - 1, z), \\ \text{If not}(y > 0) \text{ then } \mathrm{LOOP}^1_x(x, y, z) = x \ . \end{cases}$$

$$\begin{cases} \text{If } y > 0 \text{ then } \mathrm{LOOP}^1_y(x, y, z) = \mathrm{LOOP}^1_y(x + z, y - 1, z), \\ \text{If not}(y > 0) \text{ then } \mathrm{LOOP}^1_y(x, y, z) = y \ . \end{cases}$$

[4] While Closure.

Listing 3. A loop.

```
int f() {
int x,y,z;

x=1;
y=2;
z=3;

while(y>0) {
  x=x+z;
  y=y−1;
  }
...
}
```

In addition, the pairs

$$(x, \text{Loop}(1, x, \textit{current state of variables}))$$

and

$$(y, \text{Loop}(1, y, \textit{current state of variables})),$$

one for each variable modified in the loop, are inserted in the environment of function f to reflect the new state of these variables. The *current state of variables* refers to the state of the variables just before the while statement. This is just like replacing the loop in function f by the function calls: $x = \text{LOOP}_x^1(1, 2, 3)$ and $y = \text{LOOP}_y^1(1, 2, 3)$.

Rules. A rewriting system is a set of rewriting rules. A rewriting rule is composed of a left and a right part and indicates that the right part can be substituted to the left part in any term where the left part — possibly with a substitution of its variables — appears. The rules are divided into rules representing the C-- language semantics and syntax manipulating rules — mainly lists manipulation. The complete set of rules can be seen in Appendix B, but some rules are briefly commented here.

Among the rules describing the language operational semantics, we find:

- *GE* rules to translate the behavior of the *sequence* instruction;
- *Comp* rules to evaluate a new statement in the current environment;
  - An *assignment* or a *return* statement adds a new pair whose value is updated to reflect the current variables state — *Update_env* rules.
  - An *if* statement divides the environment into two parts through a *Choice* term. Each part is included in a *Branch* term and contains an *if* alternative depending on whether the condition is valid or no.

- A *while* statement creates a *WC* term and adds new pairs to the environment for the variables modified in the loop body as explained in the previous section (Sec. 2.2).

– two more *Comp* rules to tell that the statements following an *if* statement must be executed whatever alternative is chosen;
– *Branch* rules to group together two successive *if* statements by merging the conditions.

Among the syntax manipulating rules, we find:

– *Merge_env* and *Merge_L_var* to merge two lists;
– *Insert_pair* and *Insert_var* to add a pair or a variable to a list;
– *GLOV*, *GLOMV* and *GLOE* to run through a list and build a new list by extracting, respectively, variables, modified variables or expressions from the initial list.

In order to be sure that every C-- program always rewrites in a unique normal form, we must prove that the rewriting system is convergent, that is terminating — the rewriting process eventually ends — and confluent — whenever two rules can be applied to the same term, the result is identical after some rewritings whichever rule was applied. This last property can be shown with the help of the Knuth–Bendix completion algorithm [5]. An implementation of this algorithm exists in RRL the *Rewrite Rule Laboratory* [4]. We used RRL to exhibit a *lexicographic path order* relation over the symbols of the rewriting system — which guarantees it is terminating — and to apply successfully the completion algorithm.

2.3 Equation Generation

`Input` : An environment.
`Output` : A set of equations.

Only a few elements in environments will generate equations: they are the equation generators. The third and final step of the axiomatization process refines environments, extracts equation generators from environments and generates corresponding equations.

First, environment are made clearer through evaluation of the following terms:

– *Subst* terms. Subst(x, exp_1, exp_2) denotes the substitution of variable x by expression exp_2 in expression exp_1. The substitution is simply applied.
– *EP* terms. They are not necessary anymore since the distinction between effective and formal parameter is only needed for substitutions. EP(x) is replaced by x.
– *Loop* terms. They undergo a purely syntactic transformation.

$$\text{Loop}(num, variable, \{exp_1, \ldots, exp_n\}) \text{ is replaced by}$$
$$\text{LOOP}^{num}_{variable}(exp_1, \ldots, exp_n) \ .$$

Then equation generators are transformed into equations. The equation generators are:

- *lists of pairs*. They represent the state of the variables at the end of the computation. But, only the function return value is of interest, therefore, only the pair containing the function name will generate an equation.

  $\texttt{Generator}$: $\mathrm{Pair}(\dots) \cdot \dots \cdot \mathrm{Pair}(\dots) \cdot \mathrm{Pair}(\mathit{function\_name}, \mathit{expression})$.
  $\texttt{Equation}$: $\mathit{function\_name} = \mathit{expression}$.

- *Branch* terms. They appear because of an *if* statement and represent an alternative. They link a condition and a list of pairs. Again, only the pair with the function name is of interest. Each *Branch* term generates one conditional equations.

  $\texttt{Generator}$:
  $$\mathrm{Branch}\begin{pmatrix}\mathit{condition},\\ \mathrm{Pair}(\dots) \cdot \dots \cdot \mathrm{Pair}(\dots) \cdot \mathrm{Pair}(\mathit{function\_name}, \mathit{expression})\end{pmatrix} .$$

  $\texttt{Equation}$: $\mathit{condition} = \mathit{True} \Rightarrow \mathit{function\_name} = \mathit{expression}$.

- *WC* terms. They generate a family of conditional equations that defines recursively the loop functions — one loop function for each modified variable in the loop body. Two equations are needed, one for the recursive call — with the variables state modified according to the loop body — and one for the exit case which gives the result of the loop function, that is the current value of the considered modified variable.

  $\texttt{Generator}$:
  $$\mathrm{WC}\,(\mathit{num}, \mathit{cond}, \mathrm{Pair}(v_1, \mathit{exp}_1) \cdot \dots \cdot \mathrm{Pair}(v_n, \mathit{exp}_n), \{v_1, \dots, v_n\}) \ .$$

  $\texttt{Equation}$:
  $$\bigcup_{1 \le i \le m} \left\{ \begin{array}{l} \mathit{cond} = \mathit{True} \Rightarrow \\ \qquad \mathrm{LOOP}^{\mathit{num}}_{v_{\mathrm{mod}_i}}(v_1, \dots, v_n) = \mathrm{LOOP}^{\mathit{num}}_{v_{\mathrm{mod}_i}}(\mathit{exp}_1, \dots, \mathit{exp}_n) \ , \\ \mathit{cond} = \mathit{False} \Rightarrow \\ \qquad \mathrm{LOOP}^{\mathit{num}}_{v_{\mathrm{mod}_i}}(v_1, \dots, v_n) = v_{\mathrm{mod}_i} \end{array} \right\} .$$

  Here, $v_{\mathrm{mod}_1}, \dots, v_{\mathrm{mod}_m}$ are the variables modified in the loop body and $v_1, \dots, v_n$ are all the variables appearing in the C-- function. A variable v is known to have been modified when its value differs from $\mathrm{EP}(v)$ which is the value assigned to it before getting in the loop.

3 Extended Example

This section goes over the axiomatization process again, showing how the three steps of the process apply to a case study. The C-- version of the *list inversion* will serve as support (see Appendix A for some other examples).

Listing 4. Reverse function.

```
list reverse(list L) {
  list W=NULL;

  while(L != NULL) {
    W=cons(getHead(L), W);
    L=getQueue(L);
  }

  return w;
}
```

3.1 Term Generation

The term in Fig. 4 is constructed by the *term generation* step from the C-- code of listing 4.

```
GE(
  Cons_inst(Cons_inst(Cons_inst(
        End_inst,
        Assign( W, NULL) ),
        While( 1, (L<>NULL), Cons_inst(Cons_inst(
                                End_inst,
                                Assign( W, cons(getHead(L), W)) ),
                                Assign(L, getQueue(L)) ) ) ),
        Return(reverse(L), W) ),
  Cons_env(Pair(L, EP(L)),
           Empty_env ) )
```

Fig. 4. *Reverse* function initial term.

The *GE* term can be identified with its list of instructions and initial environment. The list of instructions is made up of:

- an *assignment* which comes from the initialization of the variable W at the time of its declaration;
- a *while* instruction which includes its own instruction list — two assignments corresponding to the body loop;
- a *return* instruction where the function name appears.

The environment is initially composed of one pair, $\mathrm{Pair}(L, \mathrm{EP}(L))$, associating variable L and the value L will take at the function call — its effective value denoted $\mathrm{EP}(L)$.

3.2 Environment Generation

At the *environment generation* step, the initial term of the *reverse* function is rewritten using the rewriting system in a final environment. Fig. 5 presents the environment obtained from the term of Fig. 4 once refined — as explained in Sec. 2.3 — for readability purpose.

```
Cons_env(WC(1, (L<>NULL), Cons_env(Pair(W, cons(getHead(L), W)),
                                   Cons_env(Pair(L, getQueue(L)),
                                            Empty_env ) ),
                                   Cons_var(L,
                                   Cons_var( W,
                                            End_var ) ) ),

Cons_env(Pair(L, Loop(1, L, Cons_exp(L,
                                Cons_exp(NULL,
                                         End_exp ) ) ) ),

Cons_env(Pair(W, Loop(1, W, Cons_exp(L,
                                Cons_exp(NULL,
                                         End_exp ) ) ) ),

Cons_env(Pair(reverse(L), Loop(1, W, Cons_exp(L,
                                         Cons_exp(NULL,
                                                  End_exp ) ) ) ),
Empty_env ) ) ) )
```

Fig. 5. *Reverse* function refined environment.

The environment is made up of:

- a WC term. The number 1 is attributed to the loop. The environment represents the state of all the *reverse* function variables, initialized to their effective value, after one execution of the loop body. The last WC parameter is the list of all the function variables.
- a *list of pairs*. These pairs represent the state of the function variables once the function has been executed. L and W are assigned the result to a call to a *Loop* function which will be defined during the equation generation step thanks to the information enclosed in the WC term. The last pair is the function result.

3.3 Equation Generation

The environment is made up of the following *equation generators*:

- the WC term which generates the *Loop* equations. Since L and W are modified in the loop body, two *Loop* functions are created. They are LOOP^1_L and LOOP^1_W.

– the *pair* containing the function name which equals a call to *reverse* to a call to LOOP^1_W.

Finally, *equation generation* step gives the equations of Fig. 6.

$$
\begin{aligned}
(L <> NULL) = True &=> \mathrm{LOOP}^1_W(L, W) = \\
&\quad \mathrm{LOOP}^1_W(\mathrm{getQueue}(L), \mathrm{cons}(\mathrm{getHead}(L), W)), \\
(L <> NULL) = False &=> \mathrm{LOOP}^1_W(L, W) = W, \\
(L <> NULL) = True &=> \mathrm{LOOP}^1_L(L, W) = \\
&\quad \mathrm{LOOP}^1_L(\mathrm{getQueue}(L), \mathrm{cons}(\mathrm{getHead}(L), W)), \\
(L <> NULL) = False &=> \mathrm{LOOP}^1_L(L, W) = L, \\
\mathrm{reverse}(L) &= \mathrm{LOOP}^1_W(L, NULL) \ .
\end{aligned}
$$

Fig. 6. *Reverse* function equations.

4 Related Work

Imperative languages are widely used in the industrial world which expresses a strong need for simple and user-friendly specification and verification tools. Several approaches address this challenge. We can distinguish those which generate code from specifications, from those which work with source code as raw material.

- *Program synthesis* uses a formal and high-level language to describe program specifications. The specification language semantics is well-defined enough to produce source code in various programming languages. Systems based on this approach mainly differ on the specification language which is often tuned for a particular type of application. Examples of such systems are COGITO [12], SPECWARE [10]. This approach suffers from several drawbacks. The specification language can help in saying *what* a program must do, but the language is often not sufficient to express *how* it should be done. The generated code is not as efficient as the one a programmer would produce. In addition, these systems can not be used to verify existing programs or for maintenance purpose.
- The second category of verification systems can also be divided into two sub-categories.
  - *Program annotation* requires that the user inserts program specifications in the form of annotations directly into source code. These annotations will help the system to conduct the proof (see [8] and [1] for instance).
  - *Specification generation* attempts to extract specifications from source code and verify them against user specifications. This kind of system needs no user interaction but, possibly, for the proof step. The method

exposed in [13] uses a set of well-known semantics-preserving transformations to extract specifications. Specifications are then written in a language mixing high-level and low-level content. PESCA [11] is close to our approach. This system uses algebraic semantics for the specification part and a basic imperative language for the programming part. The proofs are conducted in the LARCH PROVER [9] theorem prover. The main differences with our work come from the restrictions applied to the programming language (no loops allowed) and the method used to generate the specifications.

5 Conclusion

In this paper we have discussed a method to automatically obtain an equivalent equational formulation of a C-- program from source code. The process leading to the equations requires three steps. The central point of the discussed method is the generation of an environment by means of a rewriting system which implements the operational semantics of the C-- language. The first stage consists in building a term suitable for rewriting through the syntactic analysis of the program code. The last stage consists in translating the environment into equations.

An implementation of this method has been carried out in Java. JavaCC [5], a parser and scanner generator, has been used for the term generation step. We developed a Java version of a generic rewriting algorithm. The rewriting rules are loaded separately from a file so as to elaborate the rules with ease.

A lot of work has still to be done. Future work includes:

- adding functionalities to the C-- language in order to come closer to real imperative languages;
- implementing interfaces towards proof systems, that is, providing the equations in the specific system syntax;
- experimenting on a larger scale proving properties from the equations in proof systems. This in order to identify a class of properties and programs that can be proven sound using our method.

References

1. S. Antoy and J. Gannon. Using Term Rewriting to Verify Software. *IEEE Transactions on Software Engineering*, 20(4):259–274, 1994.
2. B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Mu noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997. http://pauillac.inria.fr/coq/coq-fra.html.
3. Fédèle C. and Kounalis E. Automatic proofs of properties of simple $C--$ modules. In *14th IEEE International Conference on Automated Software Engineering*, October 1999.

[5] Java Compiler Compiler, Metamata.

4. Kapur D. and Zhang H. *RRL : Rewrite Rule Laboratory*, May 1989.
5. Knuth D.E. and Bendix P.B. Simple word problems in universal algebras. *Computational Problems in Abstract Algebra*, 1970. Pergamon Press.
6. Kounalis E. and Urso P. Generalization discovery for proofs by induction in conditional theories. In *Proceedings of 12th International FLAIRS*. AAAI Press, 1999.
7. Baader F. and Nipkow T. *Term Rewriting and All That*. Cambridge University Press, 1998.
8. Jean-Christophe Filliâtre. Proof of imperative programs in type theory. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, volume 1657 of *Lecture Notes in Computer Science*, page 78. Springer-Verlag, March 1998.
9. Stephen J. Gar and John V. Guttag. A guide to LP, the larch prover. Technical Report 82, Digital Equipment Corporation, Systems Research Centre, 31 December 1991.
10. R. Juellig, Y. Srinivas, and J. Liu. SPECWARE: An advanced environment for the formal development of complex software systems. *Lecture Notes in Computer Science*, 1101:551, 1996.
11. Daniel Schweizer and Christoph Denzler. Verifying the specification-to-code correspondence for abstract data types. In M. Dal Cin, C. Meadows, and W. Sanders, editors, *Dependable Computing for Critical Applications 6*, volume 11 of *Dependable Computing and Fault-Tolerant Systems*. IEEE Computer Society Press, 1997.
12. O. Traynor, D. Hazel, P. Kearney, A. Martin, R. Nickson, and L. Wildman. The Cogito development system. In Michael Johnson, editor, *Algebraic Methodology and Software Technology (AMAST), Berlin*, volume 1349 of *LNCS*, pages 586–591. Springer-Verlag, December 1997.
13. Martin Ward. Abstracting a specification from code. *Journal of Software Maintenance: Research and Practice*, 5(2):101–122, June 1993.

A Examples

This appendix presents some examples of C-- programs and the equations produced by the axiomatization process.

A.1 Insertion Sort

```
list ins(int e, list L) {
list ret;

if(L==NULL)
        ret=cons(e, NULL);
else if(e<=getHead(L))
        ret=cons(e, L);
else {
        ret=getQueue(L);
        ret=ins(e, ret);
        ret=cons(getHead(L), ret);
        }
```

```
return ret;
}

list ISort(list L) {
list ret;

if(L==NULL)
        ret=NULL;
else
        ret=ins(getHead(L), IS(getQueue(L)));

return ret;
}
```

$$
\begin{array}{ll}
L = NULL & \Rightarrow \mathrm{ins}(e, L) = \mathrm{cons}(e, NULL), \\
L \neq NULL \text{ and } e \leq \mathrm{getHead}(L) & \Rightarrow \mathrm{ins}(e, L) = \mathrm{cons}(e, L), \\
L \neq NULL \text{ and } e > \mathrm{getHead}(L) & \Rightarrow \mathrm{ins}(e, L) = \\
 & \quad \mathrm{cons}(\mathrm{getHead}(L), \mathrm{ins}(e, \mathrm{getQueue}(L))), \\
L = NULL & \Rightarrow \mathrm{ISort}(L) = NULL, \\
L \neq NULL & \Rightarrow \mathrm{ISort}(L) = \\
 & \quad \mathrm{ins}(\mathrm{getHead}(L), \mathrm{ISort}(\mathrm{getQueue}(L))) \ .
\end{array}
$$

A.2 Greatest Common Divisor

```
int gcd(int x, int y)
{
  while(x!=y)
  {
    if(x>y)
      x=x−y;
    else
      y=y−x;
  }
  return x;
}
```

$$
\begin{array}{ll}
x \neq y \text{ and } x > y & \Rightarrow \mathrm{LOOP}^1_y(y, x) = \mathrm{LOOP}^1_y(y, (x - y)), \\
x \neq y \text{ and } x \leq y & \Rightarrow \mathrm{LOOP}^1_y(y, x) = \mathrm{LOOP}^1_y((y - x), x), \\
x = y & \Rightarrow \mathrm{LOOP}^1_y(y, x) = y, \\
x \neq y \text{ and } x > y & \Rightarrow \mathrm{LOOP}^1_x(y, x) = \mathrm{LOOP}^1_x(y, (x - y)), \\
x \neq y \text{ and } x \leq y & \Rightarrow \mathrm{LOOP}^1_x(y, x) = \mathrm{LOOP}^1_x((y - x), x), \\
x = y & \Rightarrow \mathrm{LOOP}^1_x(y, x) = x, \\
 & \quad \mathrm{pgcd}(x, y) = \mathrm{LOOP}^1_x(y, x) \ .
\end{array}
$$

B Rewriting System

1. GE(Cons_inst(L_inst, inst), env) $\longrightarrow$ Comp(inst, GE(L_inst, env))
2. GE(End_inst, env) $\longrightarrow$ env
3. Comp(Assign(var, exp), Empty_env) $\longrightarrow$ Cons_env(Pair(var, exp), Empty_env)
4. Comp(Assign(var, exp), Cons_env(pair, env)) $\longrightarrow$
   Update_env(Pair(var, exp), Cons_env(pair, env))
5. Comp(Return(fct, exp), Cons_env(pair, env)) $\longrightarrow$
   Update_env(Pair(fct, exp), Cons_env(pair, env))
6. Comp(Return(fct, exp), Empty_env) $\longrightarrow$ Cons_env(Pair(fct, exp), Empty_env)
7. Comp(If(cond, L_inst1, L_inst2), Cons_env(pair, env)) $\longrightarrow$ Choice(
   Branch(Update_cond(cond, Cons_env(pair, env)),
   GE(L_inst1, Cons_env(pair, env))),
   Branch(not(Update_cond(cond, Cons_env(pair, env))),
   GE(L_inst2, Cons_env(pair, env))))
8. Comp(While(num, cond, L_inst), Cons_env(pair, L_pair)) $\longrightarrow$
   Cons_env(WC(num, cond, GE(L_inst, GIE[1](Cons_env(pair, L_pair))),
   GLOV[2](Cons_env(pair, L_pair))), Merge_env(GL[3](num,
   GLOMV[4](GE(L_inst, GIE(Cons_env(pair, L_inst)))),
   GLOE[5](Cons_env(pair, L_pair))), Cons_env(pair, L_pair)))
9. Comp(inst, Choice(exp1, exp2)) $\longrightarrow$ Choice(Comp(inst, exp1), Comp(inst, exp2))
10. Comp(inst, Branch(cond, env)) $\longrightarrow$ Branch(cond, Comp(inst, env))
11. Update_cond(cond, Cons_env(Pair(var, exp), env)) $\longrightarrow$
    Update_cond(Subst(var, cond, exp), env)
12. Update_cond(cond, Empty_env)$\longrightarrow$ cond
13. Update_cond(cond, Cons_env(WC(...), env)) $\longrightarrow$ Update_cond(cond, env)
14. Update_env(Pair(x, exp1), Cons_env(Pair(x, exp2), env)) $\longrightarrow$
    Update_env(Pair(x, Subst(x, exp1, exp2)), env)
15. Update_env(Pair(x, exp1), Cons_env(Pair(y, exp2), env)) $\longrightarrow$
    Cons_env(Pair(y, exp2), Update_env(Pair(x, Subst(y, exp1, exp2)), env))
    if not equal(x, y)
16. Update_env(Pair(var, exp), Empty_env) $\longrightarrow$ Cons_env(Pair(var, exp), Empty_env)
17. Update_env(Pair(var, exp1), Cons_env(WC(...), env)) $\longrightarrow$
    Cons_env(WC(...), Update_env(Pair(var, exp1), env))

---

[1] Generate Initial Environment.
[2] Generate List Of Variables.
[3] Generate Loops.
[4] Generate List Of Modified Variables.
[5] Generate List Of Expressions.

18. Branch(cond, Choice(env1, env2)) ⟶
    Choice(Branch(cond, env1), Branch(cond, env2))
19. Branch(cond1, Branch(cond2, env)) ⟶ Branch(and(cond1, cond2), env)
20. GIE(Empty_env) ⟶ Empty_env
21. GIE(Cons_env(Pair(var, exp), env)) ⟶ Cons_env(Pair(var, EA(var)), GIE(env))
22. GIE(Cons_env(WC(...), env)) ⟶ GIE(env)
23. GLOV(Empty_env) ⟶ End_var
24. GLOV(Cons_env(Pair(var, exp), env)) ⟶ Cons_var(var, GLOV(env))
25. GLOV(Cons_env(WC(...), env)) ⟶ GLOV(env)
26. GLOMV(Empty_env) ⟶ End_var
27. GLOMV(Cons_env(Pair(var, EA(var)), env)) ⟶ GLOMV(env)
28. GLOMV(Cons_env(Pair(var, exp), env)) ⟶
    Cons_var(var, GLOMV(env)) if not equal(EA(var), exp)
29. GLOMV(Cons_env(WC(...), env)) ⟶ GLOMV(env)
30. GLOMV(Branch(cond, env)) ⟶ GLOMV(env)
31. GLOMV(Choice(env1, env2)) ⟶ Merge_L_var(GLOMV(env1), GLOMV(env2))
32. Merge_L_var(Cons_var(x, L_var1), L_var2) ⟶
    Insert_var(x, Merge_L_var(L_var1, L_var2))
33. Merge_L_var(End_var, L_var) ⟶ L_var
34. Insert_var(x, Cons_var(x, L_var)) ⟶ Cons_var(x, L_var)
35. Insert_var(x, Cons_var(y, L_var)) ⟶
    Cons_var(y, Insert_var(x, L_var)) if not equal(x, y)
36. Insert_var(x, End_var) ⟶ Cons_var(x, End_var)
37. GLOE(Empty_env) ⟶ End_exp
38. GLOE(Cons_env(Pair(var, exp), env)) ⟶ Cons_exp(exp, GLOE(env))
39. GLOE(Cons_env(WC(...), env)) ⟶ GLOE(env)
40. GL(num, Cons_var(var, L_var), Cons_exp(exp, L_exp)) ⟶
    Cons_env(Pair(var, LOOP(num, var, Cons_exp(exp, L_exp))),
    GL(num, L_var, Cons_exp(exp, L_exp)))
41. GL(num, End_var, Cons_exp(exp, L_exp)) ⟶ Empty_env
42. Merge_env(Cons_env(pair, L_pair), env) ⟶
    Insert_pair(pair, Merge_env(L_pair, env))
43. Merge_env(Empty_env, env) ⟶ env
44. Insert_pair(Pair(x, exp), Cons_env(WC(...), env)) ⟶
    Cons_env(WC(...), Insert_pair(Pair(x, exp), env))
45. Insert_pair(Pair(x, exp1), Cons_env(Pair(x, exp2), env)) ⟶
    Cons_env(Pair(x, exp1), env)
46. Insert_pair(Pair(x, exp1), Cons_env(Pair(y, exp2), env)) ⟶
    Cons_env(Pair(y, exp2), Insert_pair(Pair(x, exp1), env)) if not equal(x, y)
47. Insert_pair(Pair(x, exp), Empty_env) ⟶ Cons_env(Pair(x, exp), Empty_env)