CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D5.2
Trusted CerCo Prototype

Version 1.0

Main Authors:
Roberto M. Amadio, Nicolas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark and Paolo Tranquilli

Project Acronym: CerCo
Project full title: Certified Complexity
Proposal/Contract no.: FP7-ICT-2009-C-243881 CerCo

Abstract The trusted CerCo Prototype is meant to be the last piece of software produced in the CerCo project. It consists of

1. A compiler from a large subset of C to the Intel HEX format for the object code of the 8051 microprocessor family. The compiler also computes the cost models for the time spent in basic blocks and the stack space used by the functions in the source code. The cost models are exposed to the user by producing an instrumented C code obtained injecting in the original source code some instructions to update three global variables that record the current clock, the current stack usage and the maximum stack usage so far.

2. A plug-in for the Frama-C Software Analyser architecture. The plug-in takes in input a C file, compiles it with the CerCo compiler and then automatically infers cost invariants for every loop and every function in the source code. The invariants can be fed by Frama-C to various theorem provers to automatically verify them. Those that are not automatically verified can be manually checked by the user using a theorem prover.

3. A wrapper that interfaces and integrates the two pieces of software above with the Frama-C Jessie plugin and with the Why3 platform.

The plug-in is fully functional and it does not need to be formally verified because it does not belong to the trusted code base of programs verified using CerCo's technology. A bug in the plug-in can just infer wrong time/space invariants and bounds that will be rejected by the automatic provers.

The compiler is currently fully functional, even if not fully certified yet. The certification will continue after the end of the project.

In this deliverable we discuss the status of the Trusted CerCo Prototype at the time of the official end of the project.

Contents

```
char a[] = {3, 2, 7, -4};
char treshold = 4;

int main() {
        char j;
        char *p = a;
        int found = 0;
        for (j=0; j < 4; j++) {
                if (*p <= treshold) { found++; }
                p++;
        }
        return found;
}
```

Figure 1: A simple C program.

1 Task

The Grant Agreement describes deliverable D5.2 as follows:

> **Trusted CerCo Prototype**: Final, fully trusted version of the system.

This report describes the state of the implementation of the Trusted CerCo Prototype at the official end of the project.

2 The Trusted CerCo Compiler

2.1 Command line interface

The Trusted CerCo Compiler is the first component of the Trusted CerCo Prototype, the second one being the Frama-C plug-in already developed in D5.1 and not modified. The Trusted CerCo compiler replaces the Untrusted CerCo Compiler already delivered in D5.1 as part of the Untrusted CerCo Prototype. The Trusted and Untrusted compilers are meant to provide the same command line interface, so that they can be easily swapped without affecting the plug-in. Actually, the two compilers share the following minimal subset of options which are sufficient to the plug-in:

```
Usage: acc [options] file...
  -a                        Add cost annotations on the source code.
  -is                       Outputs and interprets all the compilation passes,
                            showing the execution traces
  -o                        Prefix of the output files.
```

Let the file `test.c` contains the code presented in Figure 1. By calling `acc -a test.c`, the user obtains the following files:

- `test-instrumented.c` (Figure 2): the file is a copy of `test.c` obtained by adding code that keeps track of the amount of time and space used by the source code.

```
int __cost = 33;

int __stack_size = 5, __stack_size_max = 5;

void __cost_incr(int incr) {
  __cost = __cost + incr;
}

void __stack_size_incr(int incr) {
  __stack_size = __stack_size + incr;
  __stack_size_max = __stack_size_max < __stack_size ? __stack_size : __stack_size_max;
}

unsigned char a[4] = { 3, 2, 7, 252, };

unsigned char treshold = 4;

int main(void)
{
  unsigned char j;
  unsigned char *p;
  int found;
  __stack_size_incr(0);
  __cost_incr(91);
  p = a;
  found = 0;
  for (j = (unsigned char)0; (int)j < 4; j = (unsigned char)((int)j + 1)) {
    __cost_incr(76);
    if ((int)*p <= (int)treshold) {
      __cost_incr(144);
      found = found + 1;
    } else {
      __cost_incr(122);
      /*skip*/;
    }
    p = p + 1;
  }
  __cost_incr(37);
  /*skip*/;
  __stack_size_incr(-0);
  return found;
  __stack_size_incr(-0);

}
```

Figure 2: The instrumented version of the program in Figure 1.

The global variable `__cost` records the number of clock cycles used by the microprocessor. Its initial value may not be zero because we emit object code to initialize the global variables of the program. The initialization code is not part of `main` (to allow `main` to be recursive, even if it is not clear from the standard if this should be allowed or not). The initial value of `__cost` is the time spent in the initialization code. The `__cost` variable is incremented at the beginning of every basic block using the `__cost_incr` function, whose argument is the number of clock cycles that will be spent by the basic block that is beginning. Therefore the value stored in the variable is always a safe upper bound of the actual time. Moreover, the difference at the begin and end of a function of the value of the `__clock` variable is also exact. The latter statement — with several technical complications — is the one that will be eventually certified in Matita.

The global variables `__stack_size` and `__stack_size_max` are handled similarly to `__cost`. Their value is meant to represent the amount of external data memory currently in use and the maximum amount of memory used so far by the program. The two values are updated by the `__stack_size_incr` function at the beginning of every function body, at its end and just before every `return`. A negative increment is used to represent stack release. The initial value of the two variables may not be zero because some external data memory is used for global and static variables. Moreover, the last byte of external data memory may not be addressable in the 8051, so we avoid using it. The compiled code runs correctly only until stack overflow, which happens when the value of `__stack_size` reaches 2^{16}. It is up to the user to state and maintain the invariant that no stack overflow occurs. In case of stack overflow, the compiled code does no longer simulate the source code. Automatic provers are often able to prove the invariant in simple cases.

In order to instrument the code, all basic blocks that are hidden in the source code but that will contain code in the object code need to be made manifest. In particular, `if-then` instructions without an explicit `else` are given one (compare Figures 1 and 2).

- `test.hex`: the file is an Intel HEX representation of the object code for an 8051 microprocessor. The file can be loaded in any 8051 emulator (like the MCU 8051 IDE) for running or disassembling it. Moreover, an executable semantics (an emulator) for the 8051 is linked with the CerCo compilers and can be used to run the object code at the end of the compilation.

- `test.cerco` and `test.stack_cerco`: these two files are used to pass information from the CerCo compilers to the Frama-C plug-in and they are not interesting to the user. They just list the global variables and increment functions inserted by the compiler and used later by the plug-in to compute the cost invariants.

When the option `-is` is used, the compilers run the program after every intermediate compilation pass. The untrusted compiler only outputs the trace of (cost) observables and the final value returned by main; the trusted compiler also observes every function call and return (the stack-usage observables) and therefore allows to better track program execution. The traces observed after every pass should always be equal up to the initial observable that corresponds to the initialization of global variables. That observable is currently only emitted in back-end languages. The preservation of the traces will actually be granted by the main theorem of the formalization (the forward simulation theorem) when the proof will be completed.

The compilers also create a file for each pass with the intermediate code. However, the syntaxes used by the two compilers for the intermediate passes are not the same and we do not output yet any intermediate syntax for the assembly code. The latter can be obtained by disassembling the object code, e.g. by using the MCU 8051 IDE.

2.2 Code structure

The code of the trusted compiler is made of three different parts:

- Code extracted to OCaml from the formalization of the compiler in Matita. This is the code that will eventually be fully certified using Matita. It is fully contained in the `extracted` directory (not comprising the subdirectory `untrusted`). It translates the source C-light code to:
  - An instrumented version of the same C-light code. The instrumented version is obtained inserting cost emission statements `COST k` at the beginning of basic blocks. The emitted labels are the ones that are observed calling `acc -is`. They will be replaced in the final instrumented code with increments of the `__cost` variable.
  - The object code for the 8051 as a list of bytes to be loaded in code memory. The code is coupled with a trie over bit vectors that maps program counters to cost labels `k`. The intended semantics is that, when the current program counter is associated to `k`, the observable label `k` should be emitted during the execution of the next object code instruction. Similarly, a second trie, reminiscent of a symbol table, maps program counters to function names to emit the observables associated to stack usage.

  During the ERTL to RTL pass that deals with register allocation, the source code calls two untrusted components that we are now going to describe.

- Untrusted code called by trusted compiler (directory `extracted/untrusted`). The two main untrusted components in this directory are
  - The `Fix.ml` module by François Pottier that provides a generic algorithm to compute the least solution of a system of monotonic equations, described in the paper [**?**]. The trusted code uses this piece of code to do liveness analysis and only assumes that the module computes a pre-fix point of the system of equations provided. The performance of this piece of code is critical to the compiler and the implementation used exploits some clever programming techniques and imperative data structures that would be difficult to prove correct. Checking if the result of the computation is actually a pre-fixpoint is instead very simple to do using low computational complexity functional code only. Therefore we do not plan to prove this piece of code correct. Instead, we will certify a verifier for the results of the computation.

    Note: this module, as well as the next one, have been reused from the untrusted compiler. Therefore they have been thoroughly tested for bugs during the last year of the project.
  - The `coloring.ml` module taken from François Pottier PP compiler, used in a compiler's course for undergraduates. The module takes an interference graph (data structure implemented in module `untrusted_interference.ml`) and colors it, assigning to nodes of the interference graph either a color or the constant `Spilled`.

An heuristic is used to drive the algorithm: the caller must provide a function that returns the number of times a register is accessed, to decrease it likelihood of being spilled. To minimise bugs and reduce code size, we actually implement the heuristics in Matita and then extract the code. Therefore the untrusted code also calls back extracted code for untrusted computations.

Finally, module `compute_colouring.ml` is the one that actually computes the colouring of an interference graph of registers given the result of the liveness analysis. The code first creates the interference graph; then colours it once using real registers as colours to determine which pseudo-registers need spilling; finally it colours it again using real registers and spilling slots as colours to assign to each pseudo-register either a spilling slot or a real register.

The directory also contains a few more files that provide glue code between OCaml's data structures from the standard library (e.g. booleans and lists) and the corresponding data structures extracted from Matita. The glue is necessary to translate results back and forth between the trusted (extracted) and untrusted components, because the latter have been written using data structures from the standard library of OCaml.

- Untrusted code that drives the trusted compiler (directory `driver`). The directory contains the `acc.ml` module that implements the command line interface of the trusted compiler. It also contains or links three untrusted components which are safety critical and that enter the trusted code base of CerCo:
  - The CIL parser [**?**] that parses standard C code and simplifies it to C-light code.
  - A pretty-printer for C-light code, used to print the `-instrumented.c` file.
  - The instrumenter module (integrated with the pretty-printer for C-light code). It takes the output of the compiler and replaces every statement `COST k` (that emits the cost label `k`) with `__cost_incr(n)` where n is the cost associated to `k` in the map returned by the compiler. A similar operation is performed to update the stack-related global variables. Eventually this modules needs to be certified with the following specification: if a run of the labelled program produces the trace $\tau = k_1 \dots k_n$, the equivalent run of the instrumented program should yield the same result and be such that at the end the value of the `__cost` variable is equal to $\Sigma_{k \in \tau} K(k)$ where $K(k)$ is the lookup of the cost of k in the cost map K returned by the compiler. A similar statement is expected for stack usage.

2.3 Functional differences w.r.t. the untrusted compiler

From the user perspective, the trusted and untrusted compiler have some differences:

- The untrusted compiler put the initialization code for global variables at the beginning of main. The trusted compiler uses a separate function. Therefore only the trusted compiler allows recursive calls. To pay for the initialization code, the `__cost` variable is not always initialized to 0 in the trusted compiler, while it is always 0 in the untrusted code.
- The two compilers do not compile the code in exactly the same way, even if they adopt the same compilation scheme. Therefore the object code produced is different and the

control blocks are given different costs. On average, the code generated by the trusted compiler is about 3 times faster and may use less stack space.

- The trusted compiler is slightly slower than the untrusted one and the trusted executable semantics are also slightly slower than the untrusted ones. The only passes that at the moment are significantly much slower are the policy computation pass, which is a preliminary to the assembly, and the assembly pass. These are the passes that manipulate the largest quantity of data, because assembly programs are much larger than the corresponding Clight sources. The reason for the slowness is currently under investigation. It is likely to be due to the quality of the extracted code (see subsection 2.5).

- The back-ends of both compilers do not handle external functions because we have not implemented a linker. The trusted compiler fails during compilation, while the untrusted compiler silently turns every external function call into a `NOP`.

- The untrusted compiler had *ad hoc* options to deal with C files generated from a Lustre compiler. The *ad hoc* code simplified the C files by avoiding some calls to external functions and it was adding some debugging code to print the actual reaction time of every Lustre node. The trusted compiler does not implement any *ad hoc* Lustre option yet.

2.4 Implementation differences w.r.t. the untrusted compiler

The code of the trusted compiler greatly differs from the code of the untrusted prototype. The main architectural difference is the one of representation of back-end languages. In the trusted compiler we have adopted a unified syntax (data-structure), semantics and pretty-printing for every back-end language. In order to accommodate the differences between the original languages, the syntax and semantics have been abstracted over a number of parameters, like the type of arguments of the instructions. For example, early languages use pseudo-registers to hold data while late languages store data into real machine registers or stack locations. The unification of the languages have brought a few benefits and can potentially bring new ones in the future:

- Code size reduction and faster detection and correction of bugs.

  About the 3/4th of the code for the semantics and pretty-printing of back-end languages is shared, while 1/4th is pass-dependent. Sharing the semantics automatically implies sharing semantic properties, i.e. reducing to 1/6th the number of lemmas to be proved (6 is the number of back-end intermediate languages). Moreover, several back-end passes —a pass between two alternative semantics for RTL, the RTL to ERTL pass and the ERTL to LTL pass— transform a graph instance of the generic back-end intermediate language to another graph instance. The graph-to-graph transformation has also been generalized and parametrised over the pass-specific details. While the code saved in this way is not much, several significant lemmas are provided once and for all on the transformation. At the time this deliverable has been written, the generalized languages, semantics, transformations and relative properties take about 3,900 lines of Matita code (definitions and lemmas).

  We also benefit from the typical advantage of code sharing over cut&paste: once a bug is found and fixed, the fix immediately applies to every instance. This becomes particularly

significant for code certification, where one simple bug fix usually requires a complex work to fix the related correctness proofs.

- Some passes and several proofs can be given in greater generality, allowing more reuse.

  For example, in the untrusted prototype the LTL to LIN pass was turning a graph language into a linearised language with the very same instructions and similar semantics. In particular, the two semantics shared the same execute phase, while fetching was different. The pass consisted in performing a visit of the graph, emitting the instructions in visit order. When the visit detected a cycle, the back-link arc was represent with a new explicitly introduced `GOTO` statement.

  Obviously, the transformation just described could be applied as well to any language with a `GOTO` statement. In our formalization in Matita, we have rewritten and proved correct the translation between any two instances of the generalized back-end languages such that: 1) the fetching-related parameters of the two passes are instantiated respectively with graphs and linear operations; 2) the execute-related parameters are shared.

  Obviously, we could also prove correct the reverse translation, from a linear to a graph-based version of the same language. The two combined passes would allow to temporarily switch to a graph based representation only when a data-flow analysis over the code is required. In our compiler, for example, at the moment only the RTL to ERTL pass is based on a data flow analysis. A similar pair of translations could be also provided between one of the two representations and a static single assignment (SSA) one. As a final observation, the insertion of another graph-based language after the LTL one is now made easy: the linearisation pass needs not be redone for the new pass.

- Pass commutation and reuse. Every pass is responsible for reducing a difference between the source and target languages. For example, the RTL to ERTL pass is responsible for the parameter passing conversion, while the ERTL to LTL pass performs pseudo-registers allocation. At the moment, both CompCert and CerCo fix the relative order of the two passes in an opposite and partially arbitrary way and it is not possible to simply switch the two passes. We believe instead that it should be possible to generalize most passes in such a way that they could be freely composed in any order, also with repetitions. For example, real world compilers like GCC perform some optimizations like constant propagation multiple times, after every optimization that is likely to trigger more constant propagation. Thanks to our generalized intermediate language, we can already implement a generic constant propagation pass that can be freely applied.

2.5 Quality of the extracted code

We have extracted the Matita code of the compiler to OCaml in order to compile and execute it in an efficient way and without any need to install Matita. The implementation of code extraction for Matita has been obtained by generalizing the one of Coq over the data structures of Coq, and then instantiating the resulting code for Matita. Differences in the two calculi have also been taken in account during the generalization. Therefore we can expect the extraction procedure to be reasonably bug free, because bugs in the core of the code extraction would be likely to be detected in Coq also.

The quality of the extracted code ranges from sub-optimal to poor, depending on the part of the formalization. We analyse here the causes for the poor quality:

- Useless computations. The extraction procedure removes from the extracted code almost all of the non computational parts, replacing the ones that are not removed with code with a low complexity. However, following Coq's tradition, detection of the useless parts is not done according to the computationally expensive algorithm by Berardi [**?**, **?**]. Instead, the user decides which data structures should be assigned computation interest by declaring them in one of the `Type_i` sorts of the Calculus of (Co)Inductive Constructions. The non computational structures are declared in `Prop`, the sort of impredicative, possibly classical propositions. Every computation that produces a data structure in `Prop` is granted to be computationally irrelevant. Computations that produce data structures in `Type_i`, instead, may actually be relevant of irrelevant, even if the extraction code conservatively consider them relevant. The result consists in extracted OCaml code that computes values that will be passed to functions that do not use the result, or that will be returned to the caller that will ignore the result.

  The phenomenon is particularly visible when the dependently typed discipline is employed, which is our choice for CerCo. Under this discipline, terms can be passed to type formers. For example, the data type for back-end languages in CerCo is parametrised over the list of global variables that may be referred to by the code. Another example is the type of vectors that is parametrised over a natural which is the size of the vector, or the type of vector tries which is parametrised over the fixed height of the tree and that can be read and updated only using keys (vectors of bits) whose length is the height of the trie. Functions that compute these dependent types also have to compute the new indexes (parameters) for the types, even if this information is used only for typing. For example, appending two vectors require the computation of the length of the result vector just to type the result. In turn, this computation requires the lengths of the two vectors in input. Therefore, functions that call append have to compute the length of the vectors to append even if the lengths will actually be ignored by the extracted code of the append function.

  In the literature there are proposals for allowing the user to more accurately distinguish computational from non computational arguments of functions. The proposals introduce two different types of λ-abstractions and *ad hoc* typing rules to ensure that computationally irrelevant bound variables are not used in computationally relevant positions. An OCaml prototype that implements these ideas for Coq is available [**?**], but heavily bugged. We did not try yet to do anything along these lines in Matita. To avoid modifying the system, another approach based on the explicit use of a non computational monad has been also proposed in the literature, but it introduces many complications in the formalization and it cannot be used in every situation.

  Improvement of the code extraction to more aggressively remove irrelevant code from code extracted from Matita is left as future work. At the moment, it seems that useless computations are indeed responsible for poor performances of some parts of the extracted code. We have experimented with a few manual optimizations of the extracted code and we saw that a few minor patches already allow a 25% speed up of the assembly pass. The code released with this deliverable is the one without the manual patches to maximize reliability.

- Poor typing. A nice theoretical result is that the terms of the Calculus of Constructions (CoC), the upper-right corner of Barendregt cube, can be re-typed in System F_ω, the

corresponding typed lambda calculus without dependent types [**?**]. The calculi implemented by Coq and Matita, however, are more expressive than CoC, and several type constructions have no counterparts in System F_ω. Moreover, core OCaml does not even implement F_ω, but only the Hindley-Milner fragment of it. Therefore, in all situations listed below, code extraction is not able to type the extracted code using informative types, but it uses the super-type `Obj.magic` of OCaml — abbreviated `__` in the extracted code. The lack of more precise typing has very limited impact on the performance of the compiler OCaml code, but it makes very hard to plug the extracted code together with untrusted code. The latter needs to introduce explicit unsafe casts between the super-type and the concrete types used by instances of the generic functions. The code written in this is very error prone. For this reason we have decided to write in Matita also parts of the untrusted code of the CerCo compiler (e.g. the pretty-printing functions), in order to benefit from the type checker of Matita.

The exact situations that triggers uses of the super-type are:

1. They calculi feature a cumulative hierarchy of universes that allows to write functions that can be used both as term formers and type formers, according to the arguments that are passed to them. In System F_ω, instead, terms and types are syntactically distinct. Extracting terms according to all their possible uses may be impractical because the number of uses is exponential in the number of arguments of sort $Type_i$ with $i \geq 2$.
2. Case analysis and recursion over inhabitants of primitive inductive types can be used in types (strong elimination), which is not allowed in F_ω. In the CerCo compiler we largely exploit type formers declared in this way, for example to provide the same level of type safety achieved in the untrusted compiler via polymorphic variants [**?**]. In particular, we have terms to syntactically describe as first class citizens the large number of combinations of operand modes of object code instructions. On the instructions we provide "generic" functions that work on some combinations of the operand modes, and whose type is computed by primitive recursion on the syntactic description of the operand modes of the argument of the function.
3. Σ-types and, more generally, dependently typed records can have at the same time fields that are type declarations and fields that are terms. This situation happens all the time in CerCo because we are sticking to the dependently typed discipline and because we often generalize our data structures over the types used in them. Concretely, the generalization happens over a record containing a type — e.g. the type of (pseudo)-registers for back-end languages — some terms working on the type — e.g. functions to set/get values from (pseudo)-registers — and properties over them. In System F_ω terms and types abstractions are kept syntactically separate and there is no way to pack them in records.
4. The type of the extracted function does not belong to the Hindley-Milner fragment. Sometimes simple code transformations could be used to make the function typeable, but the increased extraction code complexity would outweigh the benefits.

We should note how other projects, in particular CompCert, have decided from the beginning to avoid dependent types to grant high quality of the extracted code and maximal control over it. Therefore, at the current state of the art of code extraction, there seems to

be a huge trade-off between extracted code quality and exploitation of advanced typing and proving techniques in the source code. In the near future, the code base of CerCo can provide an interesting example of a large formalization based on dependent types and in need of high quality of extracted code. Therefore we plan to use it as a driver and test bench for future works in the improvement of code extraction. In particular, we are planning to study the following improvements to the code extraction of Matita:

- We already have a prototype that extracts code from Matita to GHC plus several extensions that allow GHC to use a very large subset of System F_ω. However, the prototype is not fully functional yet because we still need to solve at the theoretical level a problem of interaction between F_ω types and strong elimination. Roughly speaking, the two branches of a strong elimination always admit a most general unifier in Hindley-Milner plus the super-type `Obj.magic`, but the same property is lost for F_ω. As a consequence, we loose modularity in code extraction and we need to figure out static analysis techniques to reduce the impact of the loss of modularity.

- The two most recent versions of OCaml have introduced first class modules, which are exactly the feature needed for extracting code that uses records containing both types and term declarations. However, the syntax required for first class modules is extremely cumbersome and it requires the explicit introduction of type expressions to make manifest the type declaration/definition fields of the module. This complicates code extraction with the needs of performing some computations at extraction time, which are not in the tradition of code extraction. Moreover, the actual performance of OCaml code that uses first class modules heavily is unknown to us. We plan to experiment with first class modules for extraction very soon.

- Algebraic data types are generalized to families of algebraic data types in the calculi implemented by Coq and Matita, even if the two generalizations are different. Families of algebraic data type are traditionally not supported by programming languages, but some restrictions have been recently considered under the name of Generalized Abstract Data Types (GADTs) and they are now implemented in recent versions of OCaml and GHCs. A future work is the investigation on the possibility of exploiting GADTs during code extraction.

3 The Cost-Annotating Plug-In

The functionalities of the Cost Annotating Plug-In have already been described in Deliverables D5.1 and D5.3. The plug-in interfaces with the CerCo compiler via the command line. Therefore there was no need to update the plug-in code for integration in the Trusted CerCo Prototype. Actually, it is even possible to install at the same time the untrusted and the trusted compilers. The `-cost-acc` flag of the plug-in can be used to select the executable to be used for compilation. The code of the plug-in has been modified w.r.t. D5.1 to address two issues.

On the one side, the analysis of the stack-size consumption has been integrated into it. From the user point of view, time and space cost annotations and invariants are handled in a similar way. Nevertheless, we expect automated theorem provers to face more difficulties in dealing with stack usage because elapsed time is additive, whereas what is interesting for space usage is the maximum amount of stack used, which is not additive. On the other hand,

programs produced by our compiler require more stack only at function calls. Therefore the proof obligations generated to bound the maximum stack size for non recursive programs are trivial. Most C programs, and in particular those used in time critical systems, avoid recursive functions.

On the other side, the plug-in has been updated to take advantage of the new Why3 platform.

4 The `cerco` Wrapper

The Why3 platform is a complete rewrite of the old Why2 one. The update has triggered several additional passages to enable the use of the cost plug-in in conjunction with the Jessie one and the automatic and interactive theorem provers federated by the Why3 platform, mainly because the Jessie plug-in still uses Why2. These passages, which required either tedious manual commands or a complicated makefile, have prompted us to write a script wrapping all the functionalities provided by the software described in this deliverable.

```
Syntax: cerco [-ide] [-untrusted] filename.c
```

The C file provided is processed via the cost plug-in and then to the Why3 platform. The two available options command the following features.

- `-ide`: launch the Why3 interactive graphical interface for a fine-grained control on proving the synthesised program invariants. If not provided, the script will launch all available automatic theorem provers with a 5 second time-out, and just report failure or success.
- `-untrusted`: if it is installed, use the untrusted prototype rather than the trusted one (which is the default behaviour).

The minimum dependencies for the use of this script are

- either the trusted or the untrusted `acc` CerCo compiler;
- both Why2 and Why3;
- the cost and Jessie plus-ins.

However it is recommended to install as much Why3-compatible automatic provers as possible to maximise the effectiveness of the command. The provers provided by default were not very effective in our experience.

5 Connection with other deliverables

This deliverable represents the final milestone of the CerCo project. The software delivered links together most of the software already developed in previous deliverables, and it benefits from the studies performed in other deliverables. In particular:

- The compiler links the code extracted from the executable formal semantics of C (D3.1), machine code (D4.1), front-end intermediate languages (D3.3) and back-end intermediate languages (D4.3). The `-is` flag of the compiler invokes the semantics of every

intermediate representation of the program to be compiled. The executability of the C and machine code languages has been important to debug the the two semantics, that are part of the trusted code base of the compiler. The executability of the intermediate languages has been important during development for early detection of bugs both in the semantics and in the compiler passes. They are also useful to users to profile programs in early compilation stages to detect where the code spends more time. Those semantics, however, are not part of the trusted code base.

- The compiler links the code extracted from the CIC encoding of the Front-end (D3.2) and Back-end (D4.2). The two encodings have been partially proved correct in D3.4 (Front End Correctness Proof) and D4.4 (Back End Correctness Proof). The formal proofs to be delivered in those deliverables have not been completed. The most urgent future work after the end of the project will consist in completing those proofs.

- Debian Packages have been provided in D6.6 for the Cost-Annotating Plug-In, the Untrusted CerCo Compiler and the Trusted CerCo Compiler. The first and third installed together form a full installation of the Trusted CerCo Prototype. In order to allow interested people to test the prototype more easily, we also provided in D6.6 a Live CD based on Debian with the CerCo Packages pre-installed.