Formalizing and Verifying a Modern Build Language

Maria Christakis* [0], K. Rustan M. Leino [1], and Wolfram Schulte [2]

[0] Department of Computer Science, ETH Zurich, Switzerland
maria.christakis@inf.ethz.ch
[1] Microsoft Research, Redmond, WA, USA
leino@microsoft.com
[2] Microsoft, Redmond, WA, USA
schulte@microsoft.com

Manuscript KRML 235, 25 February 2014

Abstract. CLOUDMAKE is a software utility that automatically builds executable programs and libraries from source code—a modern MAKE utility. Its design gives rise to a number of possible optimizations, like cached builds, and the executables to be built are described using a functional programming language. This paper formally and mechanically verifies the correctness of central CLOUDMAKE algorithms.

The paper defines the CLOUDMAKE language using an operational semantics, but with a twist: the central operation ***exec*** is defined axiomatically, making it pluggable so that it can be replaced by calls to compilers, linkers, and other tools. The formalization and proofs of the central CLOUDMAKE algorithms are done entirely in DAFNY, the proof engine of which is an SMT-based program verifier.

0 Introduction

Building binary versions of software from source code is a central part of software engineering. For larger projects, this is much more involved than just invoking a compiler on a set of source files. One cares about making the process repeatable and efficient (e.g., by rebuilding only those artifacts whose sources have changed since the last build). To facilitate a good build process, it is essential to keep track of which artifacts depend on which other artifacts. A well-known utility for building software is MAKE, where the dependencies are given by users [2]. Realizing that the desired output artifact is a function of the source artifacts, the VESTA-2 system provides a functional programming language with which to describe the build recipe [3]. The correctness of the build system and any optimizations it performs is vital to the whole software development organization, so it makes sense to spend the effort required to ensure the correctness of the system.

CLOUDMAKE is a MAKE-like utility for building target artifacts from source artifacts. In this paper, we describe and formally verify the basic algorithm used by CLOUDMAKE and a key optimization it employs. Build recipes in CLOUDMAKE are, like in VESTA-2, captured by programs written in an eponymous functional programming language. The

* The work of this author was mostly done while visiting Microsoft Research.

extensible nature of CLOUDMAKE owes to a primitive operation called ***exec***, which, given a set of dependencies, invokes an external build tool to derive a set of artifacts. Because this operation is monitored by CLOUDMAKE, the range of available optimizations is greater than for MAKE. CLOUDMAKE is currently deployed at Microsoft, but it is not our intent in this paper to report on that experience. We are instead highlighting that the formalization and verification of the CLOUDMAKE algorithms is done in an industrial context since CLOUDMAKE affects a crucial part of software development at Microsoft and has a large number of users.

On the way to formally verifying the algorithms of CLOUDMAKE, our work contributes in two additional ways. First, we define CLOUDMAKE by an operational semantics, but with a twist: the extensible operation ***exec*** is described axiomatically, thus allowing a confined range of external tools to be invoked by ***exec***. We believe that other pluggable systems can be defined in a similar way. Second, the kind of tool we use for the formalization and proof is to this day still to be considered novel in light of how other semantics and optimizations have been proved (famously, cf. COMPCERT [9]): we use an SMT-based program verifier, namely DAFNY [4]. We use the functional subset of the DAFNY language to describe CLOUDMAKE's algorithms, and we state and prove theorems using *methods* (otherwise known as *procedures* or *subroutines*) with code (see, e.g., [5, 6]). In effect, this means the human verifier may provide various hints to make the proofs go through, but the human verifier never invokes any prover commands explicitly as would have been the case in an interactive proof assistant like COQ [0] or ISABELLE [12]. As a result, we perceive our tool chain as leading to a net reduction in human effort for the proof.

We proceed as follows. Sec. 1 shows the use and operation of CLOUDMAKE through a simple example. We define the formal semantics of the CLOUDMAKE language in Sec. 2, which also gives the basic algorithm and proves that it correctly allows parallel builds. We develop an optimized version of the algorithm in Sec. 3, highlighting the proof structure and typical or interesting parts of the proof. To give a sense of the effort involved in obtaining the correct theorems, we give a few statistics about the proofs in Sec. 4. The full proofs are available online[0]. We discuss related work in Sec. 5 and conclude in Sec. 6.

1 CloudMake

Syntactically, CLOUDMAKE is a purely functional subset of JAVASCRIPT. We show its abstract syntax in Fig. 0. In CLOUDMAKE, all variables are single assignment, and all global variables are evaluated on first use (whereas in JAVASCRIPT global variables are evaluated in declaration order).

We illustrate CLOUDMAKE and its potential for optimization by building a calculator. The calculator is written in C; it consists of source files `calc.c`, `add.c`, `sub.c`, and header file `num.h`, all found in the same directory. Given functions `cc` and `ln` (defined

[0] The versions as of this writing are available at `http://rise4fun.com/Dafny/n7Dm`, `http://rise4fun.com/Dafny/5iMO`, and `http://rise4fun.com/Dafny/GGnEP`, and we are maintaining any updated versions in the open-source DAFNY test suite at `http://dafny.codeplex.com`.

```
Program     ::= Stmt*
Stmt        ::= VarStmt | ReturnStmt
VarStmt     ::= var id = Expr;
ReturnStmt  ::= return Expr;
Expr        ::= Lit | id | Expr InfixOp Expr | PrefixOp Expr | Expr ? Expr : Expr
              | Expr (Expr*) | Expr .id | Expr[Expr] | LambdaExpr
Lit         ::= false | true | undefined | number | string | path | ObjLit | ArrLit
ObjLit      ::= {Binding*}
Binding     ::= id:Expr
ArrLit      ::= [Expr*]
InfixOp     ::= && | || | + | - | * | >= | ...
PrefixOp    ::= - | !
LambdaExpr  ::= id+ ⇒ Expr
```

Fig. 0. The abstract grammar of the CLOUDMAKE language, which is a subset of JAVASCRIPT. We use | to separate alternatives, * to denote 0 or more repetitions, and + to denote 1 or more repetitions; other punctuation is suggestive of the concrete syntax. Note that calls to the primitive operation ***exec*** are denoted as any other function invocations.

later) for invoking the C compiler and linker, respectively, a simple CLOUDMAKE script introduces a variable declaration for each tool call:

```
var main = ln("calc.exe", [calc, add, sub])
var calc = cc("calc.c", ["num.h"])
var add  = cc("add.c", ["num.h"])
var sub  = cc("sub.c", ["num.h"])
```

Evaluating this program consists in evaluating variable *main*. Evaluating the right-hand side of the *main* declaration requires the values of *calc*, *add*, and *sub*. Evaluating these requires evaluating *cc* on each source file, which produces the corresponding object files represented by paths *calc*, *add*, and *sub*. The derived object files are passed to the pending linker invocation in the *main* declaration, which then creates the executable *calc.exe*. While there is no internal mutable state, CLOUDMAKE *modifies external state* (the *system state*), in this case, the file system. Despite this, the evaluation in CLOUDMAKE can still be done safely in parallel, as discussed in Sec. 2.2.

Functions *cc* and *ln* are defined with calls to the primitive operation ***exec***:

```
var cc = (src, deps) ⇒ exec({ tool: "//bin/cl", args: [src],
                              deps: deps.add(src),
                              exps: [src.changeExtension(".obj")] })[0]
var ln = (exe, objs) ⇒ exec({ tool: "//bin/link", args: objs,
                              deps: objs, exps: [exe]) })[0]
```

This operation is key for the *extensibility* of CLOUDMAKE: any external tool may be invoked as part of a build (e.g., compilers, linkers, documentation generators, installers). The primitive ***exec*** takes as argument an object of the form:

```
{ tool: ..., args: ..., deps: ..., exps: ... }
```

where *tool* denotes the path of the tool to invoke, *args* are the arguments passed to the tool, *deps* are the paths of the artifacts that the tool is allowed to read, and *exps*

describe the artifacts that the tool must produce[1]. If the evaluation of ***exec*** succeeds, it returns paths to artifacts *exps* in the order specified by the argument. Note that tools like *cl* and *link* must comply with the axiomatization of ***exec*** in order to preserve the correctness of the CLOUDMAKE algorithms.

The formal semantics of CLOUDMAKE makes it possible to *reason about build specifications*. For example, we can prove that the program above has the same net effect on the system state as the following program does (where *map* is defined as usual):

```
var main = ln("calc.exe",
      ["calc.c", "add.c", "sub.c"].map(c ⇒ cc([x], ["num.h"])))
```

Moreover, CLOUDMAKE enables a number of *optimizations*, like cached, staged, incremental, and distributed builds, only the first of which is discussed in this paper. As an example of an optimization, imagine a scenario in which one builds the above calculator, modifies *calc.c*, and rebuilds. In this case, most dependency-based build systems first evaluate *main* in the above program, and then, based on computed dependencies and additional time-stamp or content-hash information, determine that (only) *calc.c* must be recompiled before the linker is called for a second time with the new *calc.obj* artifact and the *add.obj* and *sub.obj* artifacts in the cache. Instead of four tool calls, a *cached build* for this scenario requires only two such calls. Some existing build systems can be fragile when it comes to cached builds since it is easy to miss a dependency or get time stamps wrong. CLOUDMAKE uses content-based hashing for sources and fingerprints for derived artifacts, and enforces that all cached artifacts do exist in the system state. As a result, we can prove that CLOUDMAKE cached builds are equivalent to clean builds, see Sec. 3. Optimizations like this can improve performance substantially. In fact, incremental builds with caching reduce the build time of a major product shipped by Microsoft up to 100 times.

2 Formal Semantics

In this section, we define the formal semantics of CLOUDMAKE. We do so using the syntax of DAFNY, explaining its less obvious constructs as we go along. Because we do not have space to explain everything, we sometimes omit or simplify various details.

2.0 Domains

Programs The abstract syntax of CLOUDMAKE is modeled in the usual way of defining an algebraic datatype corresponding to each non-terminal in the grammar. For example, we define CLOUDMAKE's expressions in DAFNY along the following lines:

```
datatype Expr =
  exprLiteral(lit: Literal) | exprIdentifier(id: Identifier) | ...
  exprIf(cond: Expr, ifTrue: Expr, ifFalse: Expr) |
  exprInvocation(fun: Expr, args: seq⟨Expr⟩) | ...
  exprError(r: Reason)
```

[1] In the actual implementation of CLOUDMAKE, ***exec*** takes many more arguments, e.g., the current working directory, the environmental variables used by the tool, the expected return codes, etc.

In addition to the various expression forms in Fig. 0, we add a special "error" expression, which we use to signal evaluation errors.

For every datatype constructor `C`, DAFNY defines a discriminator `C?`, and the user-defined names of constructor parameters define destructors. For example, if `e` is an `Expr` and `e.exprIf?` evaluates to **`true`**, then `e` denotes a CLOUDMAKE if-then-else expression and `e.cond` denotes its guard subexpression.

Note that DAFNY builds in finite sequences, so **`seq`**$\langle\alpha\rangle$ denotes the type of sequences of elements of type α. In other places, where ordering is irrelevant, we use **`set`**$\langle\alpha\rangle$, which denotes a finite set.

For some components in the CLOUDMAKE grammar, the internal structure is irrelevant, so we simply define them as uninterpreted types:

```
type Path
type Artifact
```

System State CLOUDMAKE is a strict higher-order functional language, which can also read and write global system state during evaluation. The system state is represented as a finite map from `Path` to `Artifact`, which we roll into a record (because we will add more components of the state later on):

```
datatype State = StateCons(m: map⟨Path, Artifact⟩)
```

We define function `GetSt(p, st)` as `st.m[p]`, which returns the artifact for path `p`, and function `DomSt(st)` to return the domain of state `st`.

The system state can be written, but only in restricted ways. For one, it can only be extended—once a mapping for a path (to an artifact) has been added, it can never be changed. Also, only the ***exec*** operation can extend the state, which it does deterministically by reading some set of dependency artifacts. Abstractly speaking, from a given state A, there exists some infinite map A^* such that any state of any CLOUDMAKE program executing from A will be a finite subset of A^*. We can therefore imagine an oracle that, for a given path p and state A, tells us the artifact to which A^* maps p.

Every path in the domain of a reachable state must have received its artifact at some point, either being authored by the user or being built by the system. In the latter case, the artifact was built from other artifacts already in the state. We capture this property by saying that in a *valid* state, all the paths follow some well-founded order:

```
predicate ValidState(st: State)
{ forall p • p ∈ DomSt(st) ⟹ WellFounded(p) }
predicate WellFounded(p: Path)
```

The definition of `WellFounded` is not important until the proof of consistency of our axiomatization, see Sec. 2.3.

We now define a relation `Extends(st, st')` on states. It says that `st'` extends `st`, and that any mapping added conforms to the oracle:

```
predicate Extends(st: State, st': State) {
  DomSt(st) ⊆ DomSt(st') ∧
  (∀ p • p ∈ DomSt(st) ⟹ GetSt(p, st') = GetSt(p, st)) ∧
  (∀ p • p ∉ DomSt(st) ∧ p ∈ DomSt(st') ⟹ GetSt(p, st') = Oracle(p, st))
}
```

A property about the oracle is that state extension, which conforms to the oracle, preserves the predictions of the oracle. This is the only property of the oracle that we need for now, so we formulate it as a lemma:

```
function Oracle(p: Path, st: State): Artifact
lemma OracleProperty(p: Path, st0: State, st1: State)
  requires Extends(st0, st1);
  ensures Oracle(p, st0) = Oracle(p, st1);
```

The antecedent of the lemma is stated in a precondition (keyword **requires**) and its conclusion is stated in a postcondition (keyword **ensures**). This terminology comes from the fact that lemmas are actually methods—that is, code procedures—in DAFNY [5, 6]. The proof of the lemma would go into the method body, but we omit it for now. We will prove it once we also give a function body that defines `Oracle`.

We can now prove that `Extends` is transitive:

```
lemma Lemma_ExtendsTransitive(st0: State, st1: State, st2: State)
  requires Extends(st0, st1) ∧ Extends(st1, st2);
  ensures Extends(st0, st2);
{
  forall p { OracleProperty(p, st0, st1); }
}
```

The proof of this lemma invokes the oracle property for every path `p`. The DAFNY verifier works hard for us and supplies all other details of the proof.

2.1 Evaluation

We give the operational semantics by defining an interpreter. The central function of interest is `eval`, which reduces an expression to a value, while passing the system state. Figure 1 shows an excerpt of `eval`. It shows that literals evaluate to themselves and that, depending on the evaluation of its guard, an if-then-else evaluates to one of its arguments or to the error `rValidity`. Note that a **var** in a DAFNY expression context is simply a let binding, and the left-hand side can be a pattern like `Pair(a, b)`, which let-binds `a` and `b` such that `Pair(a, b)` equals the right-hand side.

The most interesting case is invocation. It evaluates the expression `expr.fun` and those in `expr.args`. Each such evaluation starts from the same state, `st`, and the result is a set `sts''` of next-states. Hence, for example, any side effects on the system state caused by the evaluation of `expr.fun` are not available during the evaluation of the arguments, allowing for parallelism in CLOUDMAKE. Two states are *compatible* if they map paths in their common domain to the same artifacts. A test is performed (function `Compatible`) to see if the set of next-states are compatible. If they are not, an `rCompatibility` error is returned; but if they are, the next-states are combined and, if the function denotes ***exec*** and the arguments are valid for ***exec***, then function `exec` is called.

We declare function `exec` as follows:

```
function exec(cmd: string, deps: set⟨Path⟩, exps: set⟨string⟩, st: State):
          Tuple⟨set⟨Path⟩, State⟩
```

```
function eval(expr: Expr, st: State, env: Env): Tuple⟨Expr, State⟩
  requires ValidEnv(env);
{
  if expr.exprLiteral? then
    Pair(expr, st)
  ...
  else if expr.exprIf? then
    var Pair(cond', st') := eval(expr.cond, st, env);
    if cond'.exprLiteral? ∧ cond'.lit = litTrue then
      eval(expr.ifTrue, st', env)
    else if cond'.exprLiteral? ∧ cond'.lit = litFalse then
      eval(expr.ifFalse, st', env)
    else
      Pair(exprError(rValidity), st)
  ...
  else if expr.exprInvocation? then
    var Pair(fun', st') := eval(expr.fun, st, env);
    var Pair(args', sts') := evalArgs(expr, expr.args, st, env);
    var sts'' := {st'} ∪ sts';
    if ¬Compatible(sts'') then
      Pair(exprError(rCompatibility), st)
    else
      var stCombined := Combine(sts'');
      if fun'.exprLiteral? ∧ fun'.lit.litPrimitive? then
        if fun'.lit.prim.primExec? then
          if |args'| = Arity(primExec) ∧
              ValidArgs(primExec, args', stCombined) then
            var ps := exec(args'[0].lit.str, args'[1].lit.paths,
                           args'[2].lit.strs, stCombined);
            Pair(exprLiteral(litArrOfPaths(ps.fst)), ps.snd)
          else
    ...  // various rValidity error cases
}
```

Fig. 1. Three cases from CLOUDMAKE's expression evaluation. Function `evalArgs` essentially maps `eval` over the expressions given as its second argument.

where cmd is the command to be executed (e.g., *"//bin/cl"* and its arguments), deps are the paths of all the artifacts that the command is allowed to read (e.g., *"calc.c"* and *"num.h"*), and exps (for "expectations") are the artifacts that a successful invocation of the command has to return (e.g., *"calc.obj"*). The result value contains a possibly updated state along with the set of paths to the expected artifacts (e.g., *"//derived/8208/calc.obj"*). (For brevity, we assume that all calls to ***exec*** succeed; to model the possibility of failure, exec would return an error code that eval would pass on.)

In our interpreter, we do not give function exec a body. Instead, we axiomatize the properties of exec using an unproved lemma:

```
lemma ExecProperty(cmd: string, deps: set⟨Path⟩, exps: set⟨string⟩, st: State)
  requires ValidState(st) ∧ deps ⊆ DomSt(st) ∧ Pre(cmd, deps, exps, st);
  ensures
    var Pair(paths, st') := exec(cmd, deps, exps, st);
    Extends(st, st') ∧
    (∀ e • e ∈ exps ⟹ Loc(cmd, deps, e) ∈ paths) ∧
    Post(cmd, deps, exps, Restrict(deps, st'));
```

These properties say that exec produces an extension st' of st and that the result value contains a path for every expectation. The definition of Post (not shown here) also says that those paths are in the extension. Note that ExecProperty has a precondition whereas exec does not. This is because the correctness theorem we show next only needs to consider those behaviors that emanate from this precondition.

The use of Loc requires more explanation. It determines the paths that will hold the derived artifacts. These are to be thought of as being placed in some temporary storage that is not directly accessible. The CLOUDMAKE program can use these paths as stated dependencies of other ***exec*** calls. In order for exec to be implementable, it is crucial that Loc be injective (but it need not be onto).

2.2 Race Freedom

We are now ready to show the first correctness theorem. It says that an evaluation of a CLOUDMAKE program will not result in an rCompatibility error. In other words, the compatibility test in eval will always succeed. This means that the evaluation of a function and its arguments can be done safely in parallel.

To verify in DAFNY that a *method* satisfies a (pre- and postcondition) specification, the specification is included in the signature of the method and any necessary proof hints are placed inline with the code, "intrinsically". To verify that a *function* satisfies a specification, the proof style tends to be different: typically, the specification is stated and verified as a separate lemma. We follow this "extrinsic" style here, where EvalLemma gives the property of eval to be verified. In this style, the structure of the proof of the lemma tends to mimic that of the function; in fact, sometimes it even repeats some of the computation, if for no other reason than to give names to subexpressions that are mentioned in the proof.

Figure 2 gives the race-freedom theorem as it pertains to expressions, along with an excerpt of its proof, showing the same three cases we showed for function eval in Fig. 1.

```
lemma EvalLemma(expr: Expression, st: State, env: Env)
  requires ValidState(st) ∧ ValidEnv(env);
  ensures
    var Pair(expr, st') := eval(expr, st, env);
    Extends(st, st') ∧
    (expr.exprError? ⟹ expr.r = rValidity);
{
  if expr.exprLiteral? {
  } ... else if expr.exprIf? {
    EvalLemma(expr.cond, st, env);
    var Pair(cond', st') := eval(expr.cond, st, env);
    if cond'.exprLiteral? ∧ cond'.lit = litTrue {
      EvalLemma(expr.ifTrue, st', env);
      Lemma_ExtendsTransitive(st, st', eval(expr.ifTrue, st', env).snd);
    } else if cond'.exprLiteral? ∧ cond'.lit = litFalse {
      EvalLemma(expr.ifFalse, st', env);
      Lemma_ExtendsTransitive(st, st', eval(expr.ifFalse, st', env).snd);
    } else { }
  } ... else if expr.exprInvocation? {
    EvalLemma(expr.fun, st, env);
    var Pair(fun', st') := eval(expr.fun, st, env);
    EvalArgsLemma(expr, expr.args, st, env);
    var Pair(args', sts') := evalArgs(expr, expr.args, st, env);
    var sts'' := {st'} ∪ sts';
    if Compatible(sts'') {
      var stCombined := Combine(sts'');
      Lemma_Combine(sts'', st);
      if fun'.exprLiteral? ∧ fun'.lit.litPrimitive? {
        if fun'.lit.prim.primExec? {
          if |args'| = Arity(primExec) ∧
              ValidArgs(primExec, args', stCombined) {
            var cmd, deps, exp :=
                args'[0].lit.str, args'[1].lit.paths, args'[2].lit.strs;
            ExecProperty(cmd, deps, exp, stCombined);
            var Pair(_, stExec) := exec(cmd, deps, exp, stCombined);
            Lemma_ExtendsTransitive(st, stCombined, stExec);
  ...
}
```

Fig. 2. Theorem that justifies parallel builds of the arguments to ***exec***. More precisely, the theorem shows that `eval` will never result in an `rCompatibility` error, which means that the recursive calls to `eval` do not produce conflicting artifacts, that is, do not build different artifacts for any result path.

The case for literals is trivial, so nothing needs to be done in that branch of the proof. In the case for if-then-else expressions, it is easy to see that the proof structure matches that of function `eval`. The proof invokes the induction hypothesis for the various subexpressions of the if-then-else and then uses the transitivity of `Extends` to complete the proof. Note that invoking another lemma or the induction hypothesis is just like making a (possibly recursive) call in the proof.

The proof case for ***exec*** is similar, but uses more lemmas. Not surprisingly, it also uses the axiomatized property of `exec`. Note that, other than manually spelling out the required lemma invocations, the myriad of "boring" proof details are all taken care of automatically by the DAFNY verifier.

2.3 Consistency of Axiomatization

Our proofs make use of the axiomatized properties of ***exec***. With any axiomatization, there is a risk of inadvertently introducing an inconsistency in the formalization. Therefore, we prove the existence of functions `exec`, `Oracle`, and `WellFounded` that satisfy the properties we axiomatized. We achieve this in DAFNY by introducing a *refinement module* where we give bodies to these functions and to the previously unproved lemmas we used to state axioms.

We build up the well-founded order on paths by computing well-founded *certificates*, which order the paths. (Note, these certificates, like the other things we describe in this subsection, are not part of the CLOUDMAKE algorithms; although they could in principle be built, they are used only to justify the consistency of our axiomatization.) We define our previously introduced predicate `WellFounded` to say that there exists a certificate:

```
datatype WFCertificate = Cert(p: Path, certs: set⟨WFCertificate⟩)
predicate CheckWellFounded(p: Path, cert: WFCertificate)
  decreases cert;
{
  cert.p = p ∧
  (∀ d • d ∈ LocInv_Deps(p) ⟹ ∃ c • c ∈ cert.certs ∧ c.p = d) ∧
  (∀ c • c ∈ cert.certs ⟹ CheckWellFounded(c.p, c))
}
predicate WellFounded(p: Path)
{ ∃ cert • CheckWellFounded(p, cert) }
```

Function `LocInv_Deps` gives the inverse function for the second argument of `Loc` (recall from Sec. 2.1 that `Loc` is injective). Note, DAFNY's inductive datatypes guarantee that certificates are well-founded, but the data structure itself does not provide any ordering on paths. It is the `CheckWellFounded` predicate that gives the necessary properties of paths; the certificates are used to prove the termination of the recursive calls of `CheckWellFounded`. (In a system like COQ [0] with *inductive constructions*, the predicate itself can be used as an inductive structure.)

Next, we define a function `RunTool` to model an actual tool, like a compiler, or rather, a collection of tools:

```
function RunTool(cmd: string, deps: map⟨Path, Artifact⟩, exp: string): Artifact
```

Argument cmd says which tool to invoke and exp says which of the tool's outputs we are interested in. Note that RunTool does not take the entire system state as a parameter. Instead, it takes a path-to-artifact mapping whose domain is exactly those paths that the tool invocation is allowed to depend on. By writing this as a function without a precondition, we are modeling tools that are deterministic and always return some artifact. To allow for tools that fail, perhaps because they need more dependencies than are given, we can think of RunTool sometimes as returning some designated error artifact.

We define function exec to invoke RunTool for each expectation exp in exps. The essential functionality is this:

```
var p := Loc(cmd, deps, exp);
if p ∈ DomSt(st) then st else
  SetSt(p, RunTool(cmd, Restrict(deps, st), exp), st)
```

where Restrict(deps, st) returns st with its domain restricted to deps.

Function Oracle(p, st) returns an arbitrary artifact if p is not well-founded; otherwise, it uses Skolemization (again, remember that this is for the proof only) to obtain a certificate cert for p and returns the following:

```
var cmd, deps, e := LocInv_Cmd(p), LocInv_Deps(p), LocInv_Exp(p);
RunTool(cmd, CollectDependencies(p, cert, deps, st), e)
```

where CollectDependencies recursively calls the oracle to obtain artifacts for the dependencies of p.

From these definitions, we can prove that exec does have the properties stated by ExecProperty. The proof is about 250 lines. One main lemma of the proof says that the calls above to CollectDependencies and Restrict return the same state map. A major wrinkle in the proof deals with the case when the path p given to exec already exists in the domain of the state, in which case it is necessary to prove that this is indeed what the oracle would have said.

3 Cached Builds

In this section, we formally verify the correctness of cached builds, a key optimization employed by CLOUDMAKE. This optimization effectively reduces the build times of CLOUDMAKE by making use of the fact that code changes software developers typically make between successive versions of a program are small, especially in comparison to the size of the modified program.

Cached builds enable the reuse of artifacts that have already been derived during previous, similar builds. The theorems that we show here say that cached builds are equivalent to clean builds, that is, building a program without using cached artifacts is indistinguishable from any cached build, and that, starting from any *consistent cache*, a cached build never fails due to the cache being inconsistent and the new state also has a consistent cache.

The state is now extended with a cache component represented as a hash map from paths. The cache is *consistent* when for each hashed path there exists a matching derived artifact in the system state:

```
predicate ConsistentCache(stC: State) {
  ∀ cmd, deps, e • Hash(Loc(cmd, deps, e)) ∈ DomC(stC.c) ⟹
    Loc(cmd, deps, e) ∈ DomSt(stC.m)
}
```

To verify the equivalence of cached and clean builds, we implement a wrapper around function exec described in the previous section. Specifically, the wrapper checks whether *all* expectations of a given command exist in the cache. If this is the case, it returns the paths to these expectations, otherwise it calls the previous, axiomatized version of exec to derive the expectations of the command, and then it consistently updates the cache by caching each derived expectation:

```
function execC(cmd: string, deps: set⟨Path⟩, exps: set⟨string⟩, stC: State):
           Tuple⟨set⟨Path⟩, State⟩
{
  if ∀ e | e ∈ exps • Hash(Loc(cmd, deps, e)) ∈ DomC(stC) then
    var paths := set e | e ∈ exps • Loc(cmd, deps, e);
    Pair(paths, stC)
  else
    var Pair(expr', st') := exec(cmd, deps, exps, stC);
    var stC' := UpdateC(cmd, deps, exps, st');
    Pair(expr', stC')
}
```

Note that for these proofs, we had to thread a new boolean `useCache` parameter through the definitions of the previous section and adjust the theorems proved before accordingly.

4 Proof Experience and Proof Statistics

Our file `ParallelBuilds.dfy` contains a formalization of the basic CLOUDMAKE algorithm, a proof that subexpressions of invocation expressions can be done in any order or in parallel, and a proof that the axioms used for these are consistent. Our file `CachedBuilds.dfy` contains a formalization of caches, proves again (but this time in the context of caches) that subexpressions of invocations can be done in any order or in parallel, proves a theorem that the cache handling maintains the correspondence of states, but does not again prove the consistency of axioms (which are essentially the same as before, except for the addition of the boolean `useCache` parameter that says whether or not to ignore the cache). Currently missing among the lemmas in `CachedBuilds.dfy` is a proof that the arguments of an invocation are considered valid in the cached version just when they are considered valid in the non-cached version. Finally, our file `ConsistentCache.dfy` shows that, starting from any consistent cache, a cached build never fails due to an `rInconsistentCache` error and the new state also has a consistent cache. Moreover, a consistent-cache state is reachable from any state by deleting all cache entries of the latter state.

The following table shows file sizes and verifier running times (in seconds) for the three files.

	number of lines	verification time
`ParallelBuilds.dfy`	835	237
`CachedBuilds.dfy`	1321	194
`ConsistentCache.dfy`	659	40

The times are in seconds on a 2.4 GHz laptop with eight logical cores, averaged over three runs (with a variation of less than 10 seconds among different runs). The file `CachedBuilds.dfy` is much larger because the proofs require much more manual guidance; however, we have not tried to clean up these proofs, which could make them shorter.

To develop the formalization and proofs, we used the DAFNY IDE [7] in Visual Studio, and found it to do a good job with verification-result caching and continuous background verification. The biggest annoyance we found (and saw a lot of) was time-outs. In such cases, we were not given much useful information from the verifier, and we had to wait longer (more than 10 seconds) to be given anything at all. The time-outs were mostly due to missing parts of the proof—once the proof was in place, verification times were usually low. To reduce frustrating waits, we divided up the proof in pieces—this can sometimes lead to good modularization, but in some cases it can become tedious. It seems that the proving system should be able to do such restructuring automatically and behind the scenes. To reduce the information available to the prover—in hopes of reducing the ways in which the automatic prover can get lost in its proof search—we also sometimes turned off the automatic induction and several times marked functions as "opaque", a recent feature in DAFNY that hides the definition of the functions unless the proof requests the definition to be revealed. In general, after having verified the basic algorithm used by CLOUDMAKE, we found the verification process to be incremental and require less effort.

The formalization presented in this paper has contributed to the development of CLOUDMAKE. In particular, we found parts of the English specification document for CLOUDMAKE either inadequate or more complex than necessary for our theorems to hold. Our work has led to identifying and fixing such mistakes in this document, for example in the evaluation of statements and the specification of ***exec***. Moreover, we substantially simplified the formalization for cached builds while threading the cache through our proofs.

5 Related Work

There are almost as many build systems as there are programming languages (since embedded, domain-specific build systems have been developed for almost all languages). But only a few such systems remain in active use. Here are the ones that had an impact on CLOUDMAKE. MAKE [2] introduced dependency-based builds, which are key to CLOUDMAKE's optimizations. VESTA-2 [3] used, for the first time, a functional programming language to describe dependencies, which are computed based on fingerprints instead of time stamps like in MAKE. VESTA-2 also introduced caching based on

fingerprints. Moreover, Google's build language and Facebook's BUCK[2] had an impact on CLOUDMAKE's incremental and distributed builds.

Build optimizations, akin to compiler optimizations, should be correctness preserving. However, such optimizations are typically difficult to verify since the proof must demonstrate that the semantics of the original program is equivalent to the semantics of the transformed program. Early compiler verification showed the equivalence of source and target programs with *commutative* diagrams [10] and presented the first mechanically verified compiler [11]. Other work of formally verifying the correctness of compiler optimizations was done by Lerner et al. [8]. The recent rise in the power of proof tools revitalized the area of compiler and optimizer verification. The most notable example is the COMPCERT project [9], which involved developing and proving correct a realistic compiler for a large subset of C, usable for critical embedded systems. A formal proof of correctness of function memoization has been done in the interactive proof assistant ACL2 [1].

6 Conclusion

We have formally presented and mechanically verified the central algorithms of CLOUDMAKE, a modern build language whose design allows for a number of possible optimizations. We have defined the CLOUDMAKE language using a pluggable operational semantics: the primitive operation ***exec*** is defined axiomatically and can be used to call any tool as part of a build as long as the tool complies with the axiomatization. To define the CLOUDMAKE semantics and verify its algorithms, we have used the SMT-based program verifier DAFNY. Given that CLOUDMAKE is a functional language, we have found it sufficient to use only the functional subset of the DAFNY language in our proofs. A limitation of our work is that we have not targeted verification of the CLOUDMAKE implementation, but only of its algorithms.

In the future, we plan on proving the equivalence of more optimized builds, like staged and incremental builds, to clean builds. A *staged* build uses dependency information from the last successful build to reduce the number of ***exec*** operations. Specifically, there are two stages in a staged build. First, we do a "lazy" build during which ***exec*** operations are not evaluated but are, instead, used to compute a dependency graph. For any given ***exec***, this graph shows which other ***exec*** operations must be evaluated first for the given ***exec*** to succeed, that is, which dependency artifacts of the given ***exec*** must be previously derived by other ***exec*** operations, recursively. Second, we traverse the dependency graph top-down and evaluate all the ***exec*** operations we postponed during the first stage. In practice, we only evaluate those ***exec*** operations that correspond to the changed system state between two successive builds. The main difference between staged and *incremental* builds is that during the second stage of an incremental build, the dependency graph is traversed bottom-up instead of top-down. We already have such a proof for staged builds, but we still aspire to formalize and prove the bottom-up algorithm of incremental builds, which is the optimization mostly used by CLOUDMAKE.

[2] `http://facebook.github.io/buck/`

By verifying these algorithms, we are ensuring that nothing can go wrong during such optimized builds. Our work already affects many product groups at Microsoft that rely on these optimizations to speed up the build times of large software products.

Acknowledgments We are grateful to Michał Moskal for suggestions on the proofs presented here. We also thank Valentin Wüstholz for his comments on drafts of this paper.

References

0. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
1. Robert S. Boyer and Warren A. Hunt, Jr. Function memoization and unique object representation for ACL2 functions. In *ACL2 Theorem Prover and its Applications*, pages 81–89. ACM, 2006.
2. Stuart I. Feldman. Make—A program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–265, 1979.
3. Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. *Software Configuration Management Using Vesta*. Monographs in Computer Science. Springer, 2006.
4. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
5. K. Rustan M. Leino. Automating induction with an SMT solver. In *VMCAI*, volume 7148 of *LNCS*, pages 315–331. Springer, 2012.
6. K. Rustan M. Leino. Automating theorem proving with SMT. In *ITP*, volume 7998 of *LNCS*, pages 2–16. Springer, 2013.
7. K. Rustan M. Leino and Valentin Wüstholz. The Dafny integrated development environment. In *Workshop on Formal-IDE*, 2014. To appear.
8. Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, pages 220–231. ACM, 2003.
9. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
10. John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Applied Mathematica*, volume 19 of *Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, 1967.
11. Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. In *Machine Intelligence*, volume 7, pages 51–72. Edinburgh University Press, 1972.
12. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.