Safeness of Make-based Incremental Recompilation

Jørgensen, Niels

Safeness of make-based incremental recompilation

Niels Jørgensen
Department of Informatics, Copenhagen Business School
Howitzvej 60, DK-2000 Frederiksberg, Denmark
nielsj@cbs.dk

ABSTRACT

The `make` program is widely used in the software industry to reduce compilation time in large projects. `make` skips source files that would have compiled to the same result as in the previous build. (Or so it is hoped, at least.) The crucial issue of safeness of omitting a full build-from-scratch is addressed by defining a semantic model for `make`. The model is in some ways similar to models proposed for logic programming languages, because makefiles, similarly to logic programs, have no global variables and execution is query driven. Safeness is shown to hold if a set of criteria are satisfied, including soundness, fairness, and completeness of makefile rules. The safeness result is useful for the makefile programmer because these criteria, while stated formally, are also intuitively reasonable, and may form a basis for a kind of checklist for makefile rules. The rigorous semantic definition for `make` may also be helpful in the construction of tools for automatic makefile generation.

1. INTRODUCTION

The `make` program is widely used in the software industry to reduce compilation time. It reads a makefile consisting of rules with the following meaning: "If file G is older than one or more of the files D_1, D_2, *etc.*, then execute command C", where D_1, D_2, *etc.*, are source files that G depends on, and C invokes compilation of G. This is characterized in [1] as *cascading* incremental recompilation, because recompilation spreads to other files along chains of dependency.

Historically, `make` originated [3] within the `Unix`/`C` community. It is the most useful with languages such as `C` that allow for splitting source files into implementation and interface (header) files, because then `make`'s cascading recompilation can be taylored as follows: Recompile a file either if the file itself changes, or an interface file on which it depends changes – but not merely if there is a change in the *implementation* of what is declared in the interface. This may be called conventional recompilation (cf. [13]).

Despite the widespread use of `make`, there are relatively few scientific or other publications on `make`. To my best knowledge a rigorous semantic definition has not been given in the previous work on `make` including the POSIX standardization effort [4]. A number of crucial questions – such as what are the general criteria that a rule in a makefile should satisfy ? and is incremental recompilation safe if the dependency list of a rule has been changed ? – seems to have not been adressed previously. As a consequence, in many software development projects there is uncertainty about how to use `make`. For example, the Mozilla browser project publicly states that *simultaneously* the software is built incrementally and from scratch, ".. *to make sure our dependencies are right, and out of sheer paranoia* .. " [7]. In Mozilla, incremental builds are used only to obtain a preliminary indication of the success of, say a bug fix; subsequently a program built from scratch is used for verification.

Safeness of `make`-based incremental compilation, the key result of this paper, can be stated informally in terms of three builds: If in a first step the appropriate derived files have been created in an *initial build-from-scratch*, then in a second step, the result of a *make-based incremental build* is equivalent to the result of a (second) *build-from-scratch*.

The rigorous semantic definition – certainly in a modified, tutorial form – may be of interest to the makefile programmer since writing correct makefiles by hand is difficult, and existing tools only automate standard tasks such as the generation of rules for `C` files.

A clear definition of basic `make` execution is also of interest to the designer of `make`-related tools, *e.g.* tools for generation of rules for new languages such as `Java`. (This was the starting point for the research reported in this paper). Using `make` with languages such as `Java` where compilers are available with built-in features for incremental compilation, is of interest in the following kinds of projects: First, `make` is useful if there are chains of dependencies due to compilation in multiple steps, analogously to the conventional use of `make` for `C` source files created by the parser-generator `yacc`. Second, in a build system that writes, *e.g.* configuration information to log files, `make` is useful because compilation of a file is invoked explicitly in the commands of rules, as opposed to automatically inside a compiler.

Finally, projects such as Mozilla whose development model is based on frequent building may save time if makefiles can be verified to meet the criteria for safeness. Such verification may employ abstract evaluation of makefile rules, as in abstract interpretation [2].

The analysis framework comprises a semantic definition for `make` which is in some ways similar to the semantic definition for (constraint) logic programs given in [6]. Makefile execution bears resemblence with logic program execution (as defined in, *e.g.*, [5]). Makefile execution is query-driven, and does not assign values to global variables. This similarity with logic programs also motivates the use of the notions of satisfiability of makefile rules and derivability of makefile targets.

Our analysis framework additionally comprises a small machinery for reasoning about the time-last-modified of files, since coarse-grained `make` controls recompilation on the basis of the time-last-modified of files. In contrast, more fine-grained variants of incremental compilation, including those studied in [13; 1; 11], distinguish between what kind of changes are made to source files.

Other related work includes scientific [3; 14] and tutorial [8; 12; 9] publications on `make` and makefile generators such as `mkmf` and `makedepend`. In restrospect it can be seen that Stuart Feldman's original paper on `make` [3] tacitly assumed that makefile rules satisfy properties that guarantee safeness. Walden [14] pointed to errors in makefile generators for `C` and related tools, in particular in rules whose dependencies were derived files (as opposed to source files), and proposed new algorithms for such generators.

The preliminary Sections 2-5 define notation, the subset of makefile syntax accounted for, the reference notion of a build-from-scratch, and the command execution model.

The semantic definitions in Sections 6-7 begin with rule satisfiability, whereafter operational semantics is defined in terms of specifying the order in which rules are fired if they are not satisfied.

Section 8 defines the notion of a build rule in terms of idempotence and other criteria, and subject to these criteria Sections 9-11 show soundness, completeness, and safeness of `make` execution. Section 12 concludes.

2. NOTATION

Given sets D and E we write $D \rightarrow E$ for the set of functions from D to E, $D \times E$ for the cartesian product of D and E, $\wp\, D$ for the power set of D, and D^* for the set of finite sequences of elements of D. We write *nil* for the empty list, and $E \in L$ if E occurs in the list L. The concatenation of lists $ds, ds' \in D^*$ is written $ds; ds'$.

Functions are defined in curried from, *i.e.*, having only a single argument. The function space $D \rightarrow (E \rightarrow F)$ is written as $D \rightarrow E \rightarrow F$. For a given function $F : D \rightarrow E \rightarrow E$, the symbol $\Sigma\, F$ is used for brevity to denote the function which has range $D^* \rightarrow E \rightarrow E$ and is defined as the following recursive application of F:

$$\begin{aligned} \Sigma\, F\, nil\, e &= e \\ \Sigma\, F\, (d; ds)\, e &= \Sigma\, F\, ds\, (F\, d\, e) \end{aligned}$$

3. SYNTAX OF MAKEFILES

The semantic definition is for the following subset of the language defined for `make` in the POSIX standard [4]:

The basic syntactic categories are $Name$ and $Command$. $Command$ contains a trivial command *nil*. A rule $R \in Rule$ is of the form

$$T : Ds; C$$

and contains a target $T \in Name$, a (possibly empty) list of dependencies $Ds \in Name^*$, and a command $C \in Command$.

```
pgm:  codegen.o parser.o library      # R_pgm
   cc codegen.o parser.o library -o pgm
codegen.o:  codegen.c definitions     # R_codegen.o
   cc -c codegen.c
parser.o:  parser.c definitions       # R_parser.o
   cc -c parser.c
parser.c:  parser.y                   # R_parser.c
   yacc parser.y
   mv y.tab.c parser.c
```

Figure 1: A makefile directing the compilation of a C program `pgm`. The four target rules are named $R_{\texttt{pgm}}$, $R_{\texttt{codegen.o}}$, etc.

A rule is said to define its targets. A makefile $M \in Makefile$ is a finite set of rules no two of which define the same target. An invokation of `make` comprises a makefile and a target list. If the list contains exactly one element, we call it the *initial* target.

Macro rules are omitted; this is without loss of generality because macro rules are expanded in a preprocessing phase, leaving only target rules. Multiple rules defining the same target are omitted; they can be rewritten into a single, and semantically equivalent rule. Finally, we can add a rule $(default : T; nil)$, where `T` is the first target defined in the makefile, to account for the target defaulted to when the target list of an invokation is empty.

No restrictions are imposed on what commands may occur in a rule.

Among the syntactical constructs not captured are suffix rules, the special target/dependency separator "::", and flags and options specified in `make` invokations.

A *derived* file $G \in der\, M$ is a target occurring and M and which is defined by some rule. A *source* file $G \in src\, M$ as any other target occurring in M. Finally, $targ\, M = (src\, M) \cup (der\, M)$.

The rules shown in Figure 1 are as in Feldman's [3] `C` compilation example. In all examples command lines are indicated by tabulation, which is by far the most common in practice. (In the semantic definitions we use the semicolon alternative to keep rules within a single line; in fact, both variations are POSIX compliant.) Rule commands in examples comply with the syntax of the `Unix` Bourne shell.

4. BUILDING FROM SCRATCH

Safeness of `make`-based incremental recompilation will be defined in Section 11 in terms of the reference notion of a build-from-scratch defined in this section. A build-from-scratch is the full, brute-force build in which everything is compiled. It is defined in terms of a given makefile, consistently with the common use of `make` to execute a build-from-scratch upon using targets such as `clobber` of Figure 3 to delete old derived files. Indeed, it is inconvenient to maintain the correct order of commands in a script file intended for sequential execution.

The command order in a build-from-scratch is derived from what is defined here as the induced make graph:

In the *make graph* $\mathcal{G} = \langle targ\, M, E\rangle$ induced by a makefile M, the set of edges E is the set of all ordered pairs $\langle T, D\rangle$ where D is a dependency in the rule in M defining T. The direction of an edge is from target to dependency. A make file is *well-formed* if the induced make graph is a directed

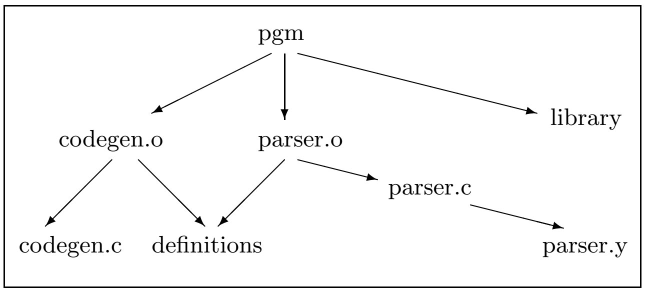

Figure 2: The make graph induced by the makefile of Figure 1.

```
all:  pgm                # $R_{\texttt{all}}$
clean:                   # $R_{\texttt{clean}}$
    rm *.o parser.c
clobber:  clean          # $R_{\texttt{clobber}}$
    rm pgm
```

Figure 3: Building from scratch is often invoked upon cleaning up with targets such as `clobber`. If added to the rules of Figure 1, $R_{\texttt{clean}}$ deletes intermediary files, and $R_{\texttt{clobber}}$ deletes all files.

acyclic graph. Figure 2 shows the make graph induced by the (well-formed) makefile of Figure 1.

We write $G \rightsquigarrow_M G'$ if there is a path from node G to node G' in M's induced make graph, possibly the trivial path from a node to itself. We write $M|_T$ for $\{(G : \mathit{Ds}; C) \in M \mid T \rightsquigarrow_M G\}$, the restriction of M to the rules "reachable" from T. Note that $\rightsquigarrow_M$ is a partial ordering on *targ* M. A sequence of commands *Cs* is *topologically ordered* with respect to a well-formed makefile M if the following holds for arbitrary rules $R = (T : \mathit{Ds}; C)$ and $R' = (T' : \mathit{Ds}'; C')$ in M: if $T \rightsquigarrow_M T'$ then C does not occur in *Cs* before C'.

Definition 1. Let M be a well-formed makefile. Then command sequence *Cs* is a *build-from-scratch* of target T with respect to M if it consists of the commands $\{C \mid (T : \mathit{Ds}; C) \in M|_T\}$ and is topologically ordered wrt. M.

Example 1. The command sequence

```
(cc -c codegen.c);
(yacc parser.y ; mv y.tab.c parser.c);
(cc -c parser.c);
(cc codegen.o parser.o library -o pgm)
```

is a build-from-scratch of target `pgm` wrt. to the makefile of Figure 1. (And so are two permutations of the sequence).

In the sequel all makefiles are assumed to be well-formed. This is consistent with the standard practice of `make` reporting a syntax error in the presence of circularly dependent targets.

5. COMMAND EXECUTION

The formal framework must capture the distinction between the contents and time-last-modified of a file. The former is the basis for defining safeness of incremental compilation, the latter is the basis for the semantic definitions, since `make` only understands time stamps and not contents.

A *file* $F \in \mathit{File} = \mathit{Time} \times \mathit{Content}$ is a pair consisting of a time-last-modified and contents. Time is equipped with a linear ordering $\Leftarrow$ and an extreme element $-\infty$ which is smaller than all other elements. Since we are not interested in the contents of a non-existing file, by abuse of notation we use $-\infty$ to denote any element $\langle -\infty, X\rangle \in \mathit{Time} \times \mathit{Contents}$.

A mapping $S \in \mathit{State} = \mathit{Name} \rightarrow \mathit{File}$ associates names with files. Changing the contents or time-last-modified of file names, whether by manual editing or command execution via `make`, or whatever, is simply modeled by a multitude of states, say a state prior to editing and another state upon editing. A mapping $S \in \mathit{State}$ is identified with its natural extension to $(\wp\, \mathit{Name}) \rightarrow \wp\,(\mathit{Time} \times \mathit{Content})$.

Command execution is modeled in terms of a function $\mathit{exec} : \mathit{Command} \rightarrow \mathit{State} \rightarrow \mathit{State}$. By abuse of notation, exec is identified with $\Sigma\, \mathit{exec}$, and thus given the command sequence $C; C'$ we write

$$\mathit{exec}\,(C; C')\; S = \mathit{exec}\, C'\; (\mathit{exec}\, C\; S)$$

Example 2. Let $S' = \mathit{exec}(\texttt{rm pgm})S$. Then $S'\texttt{pgm} = -\infty$.

Files $F, F' \in \mathit{File}$ are *equivalent*, written $F \equiv F'$, if they have the same contents, that is, if $F = \langle T, X\rangle$ and $F' = \langle T', X\rangle$. When $T \Leftarrow T'$, we write $F \Leftarrow F'$ if $F = \langle T, X\rangle$ and $F' = \langle T', X'\rangle$.

6. SATISFIABILITY

The motivation for using the logical notion of satisfiability is the intuitive, declarative reading of a makefile as a statement that certain rules must be satisfied, *i.e.*, that certain targets must be as least as new as their dependencies.

Satisfiability is defined by reversely engineering the criteria `make` uses to decide whether a rule fires. In the literature – including [3; 12; 9; 4; 14] – there is agreement about the basic criteria but, apparently, nowhere there is a fully general definition which covers all the special cases, such as the combination of an empty dependency list and a non-existing target. However, the behavior in the extreme cases can be deduced from the observable agreement concerning the semantics of "extreme" rules such as $R_{\texttt{clobber}}$ of Figure 3.

For an individual file $G \in \mathit{Name}$ we write $S \models G$ if G exists in context S, that is, if $S\, G \neq -\infty$.

Definition 2. ($\models$) Satisfiability of target rule $R = (G : \mathit{Ds}; C)$ in state S, written $S \models R$, is defined to hold if either $S \models G$ holds and additionally *Ds* is non-empty and $S \models D$ and $T \Leftarrow D$ hold for all $D \in \mathit{Ds}$; or alternatively if $C = \mathit{nil}$.

Satisfiability of makefile M in state S, written $S \models M$, is defined to hold if $S \models R$ holds for all target rules in M.

When ϕ is a file or a rule, we write $S \models \{\phi, \phi', \ldots\}$ if $S \models \phi$, $S \models \phi'$, *etc.* hold, and $S \not\models \phi$ if it does not hold that $S \models \phi$; analogously $S \not\models \{\phi, \phi', \ldots\}$ means $S \not\models \phi$, $S \not\models \phi'$,*etc.*

Example 3. Assume that M is as in Figure 1 and

$$\begin{array}{lll} S & \models & \{\texttt{codegen.c}, \texttt{definitions}, \texttt{parser.y}, \texttt{library}\} \\ S & \not\models & \{\texttt{codegen.o}, \texttt{parser.c}, \texttt{parser.o}, \texttt{pgm}\} \end{array}$$

- that is, only source files and the external library exist. Then

$$S \quad \not\models \quad \{R_{\texttt{parser.c}}, R_{\texttt{parser.o}}, R_{\texttt{codegen.o}}, R_{\texttt{pgm}}\}$$

This is because satisfiability of a rule requires that the rule's dependencies exist.

By Definition 2, rules with the trivial command *nil* constitute an exception, in the sense that they are trivially satisfied regardless of whether the dependencies are newer than the target.

Example 4. See Figure 3. For all S we have

$$\begin{array}{lcl} S & \models & R_{\texttt{all}} \\ S & \not\models & R_{\texttt{clean}} \end{array}$$

Rule $R_{\texttt{clobber}}$ is satisfied in S unless `clobber` does not exist, or `clean` does not exist, or `clean` is newer than `clobber`.

Definition 3. Target T is *up-do-date* (in S and wrt. M) if $S \models M|_T$ holds.

Thus for a target to be up-to-date is defined recursively, in terms of the target's direct and indirect dependencies. In contrast, satisfiability of a single rule is merely a relationship between the rule's target and (direct) dependencies.

Example 5. Suppose S is as in Example 3. Then `all` is not up-to-date, because `pgm` is not up-to-date. On the other hand, suppose `pgm` is up-to-date. Then `all` is up-to-date as well; many `make` implementations would in fact produce a message stating this. This is the rationale for the exception concerning rules such as $R_{\texttt{all}}$ with trivial commands.

7. OPERATIONAL SEMANTICS

The operational semantics defines what commands are executed, and in what order. We first define a semi-formal operational semantics in terms of graph theory.

Definition 4. The operational semantics of executing `make` with makefile M and initial target T is as follows:
Perform a postorder traversal of the induced make graph's derived files, starting with node T. A visit of node G, defined by rule $(G : Ds; C) \in M$ entails the following action: if $S \not\models (G : Ds; C)$ then command C is executed.

A formal semantic definition is given below in Definition 5. The format of the definition is partly in the style of denotational semantics [10], including the notational convention of using $[\![\cdot]\!]$ to distuinghish function arguments that are syntactical objects - and more generally in the attempted mathematical rigour. However, because of the finite nature of `make`'s graph traversal the full machinery of a fixpoint-based definition is not required.

The value of an expression $\mathbf{M}\,[\![M]\!]\,Ts\,S$ is the list of commands executed when `make` is invoked with initial target list Ts in the context of state S. The definition of $\mathbf{M}$ is in terms of the auxilliary functions $defines : makefile \to Name \to \{true, false\}$ and $rule : Makefile \to Name \to Rule$. $defines\ M\ T$ is $true$ if M contains a rule defining target T, in which case $rule\,M\,T$ is the unique rule defining T.

A triplet $\langle V, Cs, S\rangle \in Dom$ represents the list of nodes visited so far (V), the commands executed (Cs), and the resulting state (S). The function $\mathbf{R}$ represents rule evaluation.

Definition 5. The semantic function $\mathbf{M}$ is defined as follows.

$$\begin{array}{rcl} Dom & = & Name^* \times Command^* \times State \\ \mathbf{M} & : & Makefile \to State \to Name^* \to Command^* \\ \mathbf{T} & : & Makefile \to Name \to Dom \to Dom \\ \mathbf{R} & : & Rule \to Dom \to Dom \end{array}$$

$$\begin{array}{l} \mathbf{M}\,[\![M]\!]\ S\ Ts = \\ \qquad \text{let } \langle V, Cs, S'\rangle = \Sigma\ (\mathbf{T}\,[\![M]\!])\ Ts\ \langle nil, nil, S\rangle \text{ in } Cs \\ \mathbf{T}\,[\![M]\!]\ T\ \langle V, Cs, S\rangle = \\ \qquad \text{if } T \in V \text{ or not } defines\ M\ T \text{ then } \langle V, Cs, S\rangle \\ \qquad\ \text{else } \mathbf{R}\,[\![(rule\ M\ T)]\!]\ \langle (V; T), Cs, S\rangle \\ \mathbf{R}\,[\![T : Ds; C]\!]\ \langle V, Cs, S\rangle = \\ \qquad \text{let } \langle V', Cs', S'\rangle = \Sigma\ (\mathbf{T}\,[\![M]\!])\ Ds\ \langle V, Cs, S\rangle \text{ in} \\ \qquad\ \text{if } S' \models (T : Ds; C) \text{ then } \langle V', Cs', S'\rangle \\ \qquad\ \ \text{else } \langle V', (Cs'; C), exec\ C\ S'\rangle \end{array}$$

The definition of $\mathbf{M}$ is in some ways similar to the semantic definition for constraint logic programs given in [6], due to the similarity between the reduction of a list of goals in constraint logic program exectution, vs. `make`'s reduction of a list of targets.

In the sequel we build on the graph-based as well as the formal semantic definition.

8. DERIVABILITY

The goal in the remainder of the paper is to establish soundness, completeness, and safeness results. As a prerequisite, we define a collection of desirable properties of makefile rules, and compound them in the notion of a build rule.

First, a build rule must be sound in the sense that executing the rule's command renders the rule satisfiable.

Second, we require a rule $(T : Ds; C)$ to be *fair* in M and wrt. state S:

$$\begin{array}{cl} & fair\ (T : Ds; C)\ M\ S \\ \Leftrightarrow & \forall T' \in (der\ M) \setminus \{T\} : exec\ C\ S\ T' = S\ T' \end{array}$$

Thus, execution of a fair rule's command does not interfere with other rules, say, by updating or deleting their targets. Many makefiles contain rules with commands that write additional information to log files, recording for instance the time a build was started. Writing to such files is consistent with fairness which only precludes the "touching" of files whose names are (derived) targets of the makefile.

Soundness and fairness of rules suffice to establish soundness (see Section 9) and completeness (see Section 10) of `make`. (It is possible to relax the notion of fairness, at the expense of a more complex definition as well as more complex proofs).

Third, to infer safeness of incremental builds, we require a rule $(T : Ds; C)$ to be *complete* wrt. S:

$$\begin{array}{cl} & complete\ (T : Ds; C)\ S \\ \Leftrightarrow & \forall S' : S'\ Ds \equiv S\ Ds : exec\ C\ S'\ T \equiv exec\ C\ S\ T \end{array}$$

Thus when executing a complete rule's command, the effect on the rule's target is independent of any file not occurring in the dependency list.

Definition 6. Let M and S be given. Consider a rule $R = (T : Ds; C)$ and let $S_{Ds} = exec\,Cs\,S$, where Cs is a sequence of commands containing, for each dependency $D \in Ds$, a subsequence for the building-from-scratch of D with respect to S and M. Then R is a *build rule* in M wrt. S, written $M \vdash_S R$, if R satisfies the following:

$exec\,C\,S_{Ds} \models R$	(soundness of R)
$fair\ R\ M\ S_{Ds}$	(fairness of R wrt. other derived targets)
$complete\ R\ S_{Ds}$	(completeness of R's dependency list)

The three required properties are defined in terms of a state S_{Ds} in which all elements of the dependency list Ds of the rule in question have been created. If the dependencies are source files, that state is the same as the initial state S. If the dependencies are themselves derived targets – for example `parser.c` in Figure 1 – the relevant context is indeed a state wherein they have been created by their respective rules.

Since it is unfeasible for the human programmer to verify completeness when there are many source files, tools such as `mkmf` [8] for generating makefiles for `C` programs are of great importance. Essentially the method is to scan source files for dependent-on files, assuming compliance with suffix conventions for `C`, `yacc`, *etc.* Walden's [14] analysis showed that a number of tools did not generate (in our terminology) complete rules for targets with dependencies that were derived targets. Our notion of build rules supplements Walden's work because our notion of (rule) completeness is defined formally and independently of `C`, and so provides a stronger basis for arguing the correctness of makefile generators based on, say, the rectified algorithm proposed by Walden.

Example 6. Let makefile M consist of the rules $\{R_{\texttt{pgm}}, R_{\texttt{codegen.o}}, R_{\texttt{parser.o}}, R_{\texttt{parser.c}}, R_{\texttt{all}}, R_{\texttt{clean}}, R_{\texttt{clobber}}\}$ (cf. Figures 1 and 3). Then $R_{\texttt{all}}$, $R_{\texttt{pgm}}$, and $R_{\texttt{parser.c}}$ are build rules in M for all S. (This assumes the usual semantics of the commands `cc`, `yacc`, `rm`, *etc.*) $R_{\texttt{clean}}$ and $R_{\texttt{clobber}}$ are never build rules in M. $R_{\texttt{codegen.o}}$ and $R_{\texttt{parser.o}}$ are sound and fair, and whether they are complete depends on the contents, in a given state S, of the files *codegen.c* (for $R_{\texttt{codegen.o}}$) and *definitions* (for both rules).

Derivability of a target ($M \vdash_S T$) is defined in terms of reachable build rules, analogously to Definition 3 of a target being up-to-date ($S \models M|_T$) in terms of reachable satisfiable rules:

Definition 7. ($\vdash$) Target T is *derivable* from makefile M in S, written $M \vdash_S T$, if M defines T and $M|_T$ contains only build rules wrt. S.

The notion of derivability of targets complies with the common practice of using the notion of a derived file to refer to a file that appears as the targets of make rules and is created by executing the rule's command.

9. SOUNDNESS: MAKE BRINGS TARGETS UP-TO-DATE

Proposition 1 below can be read as a soundness result guaranteeing that execution of a makefile yields satisfiability, via changing the file state.

If a target T is derivable and all relevant source files exist, then T becomes up-to-date upon execution of `make` with target T:

PROPOSITION 1. *Assume $S \models src\ (M|_T)$ and $M \vdash_S T$, and let $S' = exec\, S\, (\mathbf{M}\, [\![M]\!]\, S\, T)$. Then $S' \models M|_T$.*

PROOF. All rules in $M|_T$ are visited. Soundness of build rules ensures that upon evaluating a rule, the rule is satisfied. Fairness ensures that a rule, once satisfied, is not rendered unsatisfied qua the execution of another rule's command. $\Box$

Since only the source files of M's restriction to T are required to exist, M can be a "real" makefile with non-build rules such as $R_{\texttt{clobber}}$ of Figure 3.

10. COMPLETENESS: MAKE INVOKES A BUILD-FROM-SCRATCH (IF REQUIRED)

Proposition 3 of this section can be read as a completeness result: `make` yields a build-from-scratch if no derived files exist.

Completeness is a consequence of the more general result expressed as Proposition 2 and which says the following. The rule defining G fires if and only if target G is reachable from the initial target and is not up-to-date. Or we may say metaphorically that recompilation cascasdes along any path from a not up-to-date target to the initial target, cf. [1].

PROPOSITION 2. *Assume $S \models src\, M$ and $M \vdash_S T$ and let $(G : Ds; C) \in M|_T$. Then*

$$C \in \mathbf{M}\, [\![M]\!]\, S\, T \Leftrightarrow S \not\models M|_G$$

PROOF. By soundness and fairness of build rules $\Box$

Example 7. Let M be as in Example 3 and suppose $S \models M$. Now assume `parser.y` is edited, yielding state S_{edited} that satisfies all rules in M except for $R_{\texttt{parser.c}}$. Then the commands belonging to $\mathbf{M}\, [\![M]\!]\, S_{edited}\, \texttt{all}$ will be the commands of the rules $R_{\texttt{pgm}}$, $R_{\texttt{parser.o}}$, and $R_{\texttt{parser.c}}$. Rule $R_{\texttt{codegen.o}}$ does not fire.

PROPOSITION 3. *Assume $S \models src\, M$, $S \not\models der\, M$, and $M \vdash_S T$ and let $(G : Ds; C) \in M|_T$. Then $\mathbf{M}\, [\![M]\!]\, S\, T$ is a build-from-scratch of T wrt. M.*

PROOF. By Proposition 2 all rules in $M|_T$ fire. Since `make` traverses the make graph in post order, they are executed in topological order, so the command sequence executed is a build-from-scratch. $\Box$

11. SAFENESS OF INCREMENTAL BUILDS

The scope of safeness is a series of states obtained in cycles of building and editing.

The build part of a cycle is essentially a pre-build state and a post-build state: A *build cycle* $\mathcal{B} = \langle S, M, T, S_{build}\rangle$ consists of a state S, a makefile M, and a target T such that $M \vdash_S T$, plus another state S_{build} satisfying $S_{build} \models M|_T$ and $S_{build} \equiv exec Cs\, S$, where Cs is a build-from-scratch of T with respect to M. That is, in the post-build state, target T is up-to-date and the contents of files are as by a build-from-scratch. Typically the first build cycle is created by actually doing a build-from-scratch; and Theorem 1 below states that in subsequent build cycles, a `make`-based, incremental build suffices.

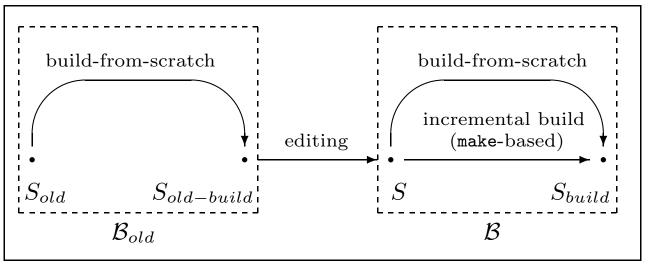

Figure 4: A sequence of states associated with build cycles $\mathcal{B}_{old}$ and $\mathcal{B}$. Arrows represent transitions between states. Safeness is illustrated by the two arrows both leading to the state S_{build}.

Build cycle $\mathcal{B} = \langle S, M, T, S_{build}\rangle$ is obtained from build cycle $\mathcal{B}_{old} = \langle S_{old}, M_{old}, T, S_{old-build}\rangle$ in an *edit cycle* if the following holds:

$$S\ (der\ M) = S_{old-build}\ (der\ M) \quad (1)$$
$$G \in (der\ M) \setminus (der\ M_{old}) \ \Rightarrow S\ G = \perp \quad (2)$$
$$(G : Ds; C) \in M \wedge (G : Ds_{old}; C_{old}) \in M_{old} \ \Rightarrow C = C_{old} \quad (3)$$

Equation (1) and implication (2) essentially limit editing to source files and the makefile: (1) states the obvious requirement that upon building in $\mathcal{B}_{old}$, editing is not allowed to change neither contents nor time-last-modified of a derived file. (2) says that if editing the makefile introduces a new derived file, then a file with that name may not already exist, which is also an obvious requirement.

Implication (3) lays down what kind of makefile editing is allowed: it rules out changing a command associated with a target defined in the makefile both before and after editing. (Indeed, upon such editing a target should always be rebuilt, and as is, make is incapable of enforcing this). While the notion of an edit cycle does not capture command editing, the notion does capture editing the dependencies of the makefile, as well as the introduction or elimination of targets in the makefile. Of course, the dependencies must be correct with respect to the editing of the source files that has taken place, in the sense required by the definition of build rules.

Figure 4 illustrates safeness of make-based incremental recompilation. The Figure shows a sequence of states, where S_{old} and $S_{old-build}$ are associated with build cycle $\mathcal{B}_{old}$, and S and S_{build} are the corresponding states in $\mathcal{B}$, which is in turn obtained from $\mathcal{B}_{old}$ in an edit cycle. The two arrows leading to the final state S_{build} illustrate safeness, which is expressed by the following Theorem.

THEOREM 1. *Suppose build cycle $\mathcal{B} = \langle S, M, T, S_{build}\rangle$ is obtained from some build cycle in an edit cycle. Define $S_{\texttt{make}} = exec\ (\mathbf{M}\ [\![M]\!]\ S\ T)\ S$. Then for any derived file $G \in (der\ M|_T)$ we have $S_{\texttt{make}}\ G = S_{build}\ G$.*

PROOF. Consider an arbitrary target G defined by a rule $R = (G : Ds; C) \in M|_T$. Assume, by induction, that $S_{\texttt{make}}\ D \equiv S_{build}\ D$ holds for any dependency $D \in Ds$ which is a derived target. It suffices to show

$$S_{\texttt{make}}\ G \equiv S_{build}\ G \quad (4)$$

Assume $S \not\models R$. Then $C \in \mathbf{M}\ [\![M]\!]\ S\ T$ (by Proposition 2), so G is re-created in the make-based build, and thus (4) holds.

Assume $S \models R$. Consider the build cycle $\mathcal{B}_{old} = \langle S_{old}, M_{old}, T, S_{old-build}\rangle$ from which $\mathcal{B}$ was obtained in an edit cycle. It must be the case that $G \in der\ (M_{old}|_T)$ (by (2)), so in the context of $S_{\texttt{make}}$, file G is as built in $\mathcal{B}_{old}$. Assumption (3) excludes command editing, so (4) holds in this case as well. □

Theorem 1 is directly applicable to a series of incremental builds; only the initial build need be a build-from-scratch.

12. CONCLUSION

The main result is Theorem 1 which states sufficient criteria for make-based, incremental recompilation to produce the same result as a build-from-scratch. The theorem allows for changing a rule's dependency list but not its command part, and for adding or deleting targets. Safeness is shown to hold subject to soundness, fairness, and completeness properties of makefile rules, as defined on the basis of a formal model of makefile execution.

Further results shown include soundness and completeness of make. The formal model also brings out the similarity with logic program execution, which motivates the use of terminology from logic, *e.g.*, satisfiability of makefile rules.

From a practical point of view, the analysis pursued here may be of interest as the basis for tutorial material, complementing available writings such as [12; 9], about what kind of safety can be achieved by make, and guidelines for writing makefiles that attain it. The properties which suffice for safeness and which are compounded in the notion of a build rule are all natural, and may be translated into rules of thumb for makefile programming, for example: Execution of the command of a build rule must render the rule satisfiable (cf. rule soundness); and execution of the command of a build rule should not effect the targets of other rules (cf. rule fairness).

Finally, verification that makefile rules satisfy the various properties is given a strong basis because of the formal approach, for example the properties are stated generically in the sense of independently of any particular programming language.

13. REFERENCES

[1] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, Vol. 3 (1), January 1994, pages 3-28.

[2] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *Proc. ACM Symposium on Artificial Intelligence and programming languages*, SIGPLAN Notices, Vol. 12 (8), 1977, pages 1-12.

[3] S. I. Feldman. Make - a program for maintaining computer programs. *Software - Practice and Experience*, Vol. 9, 1979, pages 255-265.

[4] Institute of Electrical and Electronics Engineers. *Information technology - Portable Operating System Inferface (POSIX)* . ANSI/IEEE Std. 1003.2, 1993, Part2: Shell and Utilities, Volume 1, pages 1013-1020.

[5] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, 1984.

[6] K. Marriott and H. Søndergaard. Analysis of constraint logic programs, *Proc. North American Conference on Logic Programming*, Austin, 1988, pages 521-540.

[7] The Mozilla build process is described at `http://www.mozilla.org/tinderbox.html` in the context of a presentation of the build tool "Tinderbox". The quote about paranoia is from this web page as of February 14, 2000.

[8] P.J. Nicklin. Mkmf - makefile editor. *UNIX Programmer's Manual 4.2 BSD*, June 1983.

[9] A. Oram and S. Talbott. Managing projects with make. O'Reilly, 1993.

[10] D.A. Schmidt. *Denotational semantics - a methodoogy for language development.* Allyn and Bacon, 1986.

[11] Z. Shao and A. W. Appel. Smartest Recompilation. *20th ACM Symposium on Principle of Programming Languages*, January 1993.

[12] Richard Stallman and Roland McGrath. GNU Make, Version 3.77. Free Software Foundation, 1998.

[13] W. F. Ticky. Smart recompilation. ACM Transactions on Programming Languages and Systems, Vol. 8 (3), July 1986, pages 273-291.

[14] K. Walden. Automatic Generation of Make Dependencies. *Software - Practice and Experience*, Vol. 14 (6), June 1984, pages 575-585.