# Safeness of Make-Based Incremental Recompilation

Niels Jørgensen

Department of Computer Science, Roskilde University
P.O. Box 260, DK-4000 Roskilde, Denmark
nielsj@ruc.dk

**Abstract.** The `make` program is widely used in large software projects to reduce compilation time. `make` skips source files that would have compiled to the same result as in the previous build. (Or so it is hoped.) The crucial issue of safeness of omitting a brute-force build is addressed by defining a semantic model for `make`. Safeness is shown to hold if a set of criteria are satisfied, including soundness, fairness, and completeness of makefile rules. Conditions are established under which a makefile can safely be modified by deleting, adding, or rewriting rules.

**Keywords.** Make, incremental recompilation, semantic model.

## 1 Introduction

The `make` program 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 the execution of $C$ creates $G$ by compiling the sources. This is characterized in [1] as *cascading* incremental recompilation, because recompilation spreads to other files along chains of dependency.

Safeness of `make`-based incremental compilation, the key result of this paper, can be stated as follows: Suppose we build a program brute-force, and then edit the source files, and possibly the makefile as well. Then under certain assumptions about the makefile rules and the kind of editing performed, the result of *make-based incremental recompilation* is equivalent to the result of a (second) *brute force build*. The result also applies to repeated cycles of editing and incremental recompilation.

The required properties of makefile rules are intuitively reasonable, for example, a fair rule may only create or update its own derived target. In confirming intuition about `make`, the safeness result provides formal justification for the existing practice of using `make`. Moreover, the result establishes that one may rely on `make` for incremental recompilation in situations where this is not obvious, for example upon certain modifications of the makefile.

*Comparison with related work:*

Historically, `make` originated [4] 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 instrumented as follows: Recompile a file either if the file itself changes, or an interface file on which it depends changes – but not if there is merely a change in the *implementation* of what is declared in the interface. Of course, the scheme is still extremely simple, and many files will be recompiled redundantly, for example, if a comment in a header file is modified.

A number of techniques exist for incremental recompilation which require knowledge about the syntax and semantics of the programming language being compiled. Tichy [17] coined the notion of *smart recompilation* to describe a scheme for recompilation based on analysis of the modifications made to a file, to determine whether recompilation of a file that depended on the file would be redundant, and applied the scheme to a variant of `Pascal`. A variant of smart recompilation was proposed by Elsman [3] to supplement various other techniques for compiling Standard ML programs, and incorporated into the ML Kit with Regions compiler. Syntax directed editors [2, 5] have been developed which perform compilation-as-you-type, and where the unit of granularity may be language constructs such as an individual assignment.

The level of granularity in `make`-based incremental recompilation is the file, and `make` controls the recompilation of files merely on the basis of their time stamps. Indeed, `make` is useful for tasks involving file dependencies in general (see [13] for some interesting examples) not just those that arise in the compilation of programming languages. An analysis of `make` must take a similar "blackbox" approach to files. The analysis framework comprises a small formal machinery for reasoning about execution of commands that appear in makefile rules. The machinery allows for proving the equivalence of certain command sequences, comprising the same commands but in a different order and possibly duplicated. representing brute-force vs. incremental recompilation.

The analysis framework also comprises a semantic definition for `make` which is in some ways similar to the semantic definition for (constraint) logic programs given in [8]. Makefile execution resembles logic program execution because it is query-driven and does not assign values to global variables.

Despite the widespread use of `make`, there are only few scientific or other publications on `make`. They include presentation or analysis of tools and methods [4, 18, 9], standardization [6], and tutorials [11, 12, 16] on `make` and makefile generators such as `mkmf` and `makedepend`. In retrospect it can be seen that Stuart Feldman's original paper on `make` [4] tacitly assumed that makefile rules satisfy properties that guarantee safeness. Walden's [18] analysis revealed errors in makefile generators for `C`. In the terminology of this paper, he showed that they did not generate complete rules for targets whose dependencies were derived targets. The framework supplements work such as Walden's because the notion of rule completeness is defined formally and independently of `C`.

*Contribution:*

The main contribution is the definition of criteria that makefile rules should meet as prerequisites for safeness of `make`-based recompilation, and the proof that they are sufficient when editing of sources and makefile is constrained.

There are two ways that the rigorous formulation of criteria for makefile rules may be useful:

First, the criteria may be of interest in a modified, tutorial form directed at the makefile programmer. The criteria can be restated as three rules of thumb. Writing correct makefiles by hand is difficult, and existing tools only automate standard tasks such as the generation of rules for `C` files.

Second, the criteria are also of interest in the construction and verification of `make`-related tools. For example, the starting point for this paper was an attempt to verify that the `Java` compilation rules generated by the tool `JavaDeps` [14] were appropriate, which was a practically important problem in a large software development effort I was part of. It seemed that there were no criteria against which the rules generated could be measured.

Using `make` with new languages where rule-generating tools are not mature (or available at all) may be of interest for several reasons, even for languages having compilers with built-in features for incremental recompilation. First, `make` is useful if there are chains of dependencies due to compilation in multiple steps, analogous to the conventional use of `make` for `C` source files created by the parser-generator `yacc`. Second, in a sophisticated build system that monitors the compilation process, and writes configuration information to system-specific log files, `make` is useful because compilation of a file is invoked explicitly in makefile rules, as opposed to automatically inside a compiler.

Because a number of crucial questions about `make` whose answer require a rigorous, formal approach have apparently not been addressed previously, in many software projects there is little confidence in `make`. For example, the Mozilla browser project states that it builds incrementally *and* brute force, "*.. to make sure our dependencies are right, and out of sheer paranoia ..*" [10]. In Mozilla, incremental builds are used mainly as a regression test, to see whether the browser compiles successfully; if compilation succeeds, the browser is built brute force using the exact same sources.

*Organization of the paper:*

Sections 2-5 define notation, the subset of makefile syntax accounted for, the notion of a brute-force build, and the command execution model.

The semantic definitions in Sections 6-7 comprise rule satisfiability and semantics of `make`'s execution of a makefile and an initial target.

Section 8 defines the notions of derivability and build rules in terms of rule completeness, fairness, and soundness. Section 9 contains the main result, safeness of `make`-based incremental recompilation. Section 10 discusses the validity of the `make` model, and Section 11 concludes. An appendix contains proof details.

## 2 Notation

$X \to Y$ is the set of functions from $X$ to $Y$, $X \times Y$ is the Cartesian product of $X$ and $Y$, $X^*$ is the set of finite sequences of elements of $X$, and $\wp\, X$ is the power set of $X$. $nil$ is the empty sequence. The concatenation of sequences $L$ and $L'$ is written $L; L'$. We write $X \in L$ if $X$ occurs in the sequence $L$, and $L \subseteq L'$ if $X \in L$ implies $X \in L'$, and $L' \setminus L$ for $\{X \mid X \in L' \wedge X \notin L\}$.

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

$$\Sigma\, F\, nil\, E = E$$
$$\Sigma\, F\, (D; L)\, E = \Sigma\, F\, L\, (F\, D\, E)$$

## 3 Syntax of Makefiles

For simplicity, the definitions given in this paper of syntax and semantics of makefiles are concerned only with a subset of the makefile language defined in the POSIX standard [6].

The basic syntax categories are $Name$ and $Command$. $Command$ contains a (neutral) command $nil$. A rule $R \in Rule$ is of the form

$$Ts : Ds; C$$

and contains a nonempty list of derived targets $Ts \in Name^*$, a (possibly empty) list of dependency targets $Ds \in Name^*$, and a command $C \in Command$. The rule is said to derive $Ts$, depend on $Ds$, and define $C$. A makefile $M \in Makefile$ is a finite set of rules no two of which derive the same target. An invocation of `make` comprises a makefile and an initial target. $targ\, M$ is the set of all targets occurring in $M$.

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 deriving the same target are omitted; they can be rewritten into a single, and semantically equivalent rule. Finally, an invocation of `make` with multiple or zero initial targets can be modeled by adding an extra rule to the given makefile.

Among the syntactical constructs not captured is the special separator "::" of derived targets vs. dependency targets.

The rules shown in Figure 1 are as in Feldman's [4] `C` compilation example. In all makefile examples, command lines are indicated by tabulation, which is by far the most common in practice. In the formal model, the semicolon is used instead for brevity. (Both are POSIX compliant.) In all examples, rule commands comply with the syntax of the `Unix` Bourne shell, and rules derive only a single target. Additionally, in examples it is assumed that $M$ consists of the rules listed in Figures 1 and 2, and the domain of targets $Name$ is assumed to contain only names that occur as targets in $M$ (and not names such as `cc`).

```
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
```

**Fig. 1.** A makefile for building a C program `pgm`. We refer to the rules as $R_{\texttt{pgm}}$, etc.

## 4 Commands and Files

Execution of a rule's command changes the contents and time stamps of files. The formal framework must capture the distinction between file contents (the basis for defining safeness of incremental compilation) and the time-last-modified field in a file's directory entry (which determines whether a rule fires).

A *file* $F \in File = Time \times Content$ is a pair consisting of a time stamp and contents. A mapping $S \in State = Name \to File$ associates names with files; by abuse of notation, it is identified with its natural extension to $(\wp\, Name) \to (\wp\, File)$. For a given rule $R = (Gs : Ds; C)$, the set $\{S\, T \mid T \in Gs \vee T \in Ds)\}$ is written as $S\, R$.

No ordering relation on $Time$ is required. This is because the definition of rule satisfiability to be given below (Section 6) is abstract in the sense that it does not specify *how* time stamps determine satisfiability.

Command execution is modeled in terms of a function $exec : Command \to State \to State$. The value of $exec\, nil\, S$ is $S$. The function $exec$ is identified with $\Sigma\, exec$, so the following expressions denote the same file state:

$$exec\,(C; C')\,S = \Sigma\, exec\,(C; C')\,S = exec\, C'\,(exec\, C\, S)$$

Files $F, F' \in File$ are *equivalent*, written $F \equiv F'$, if they have the same contents, that is, if $F = \langle T, X\rangle$ and $F' = \langle T', X\rangle$. Equivalence is lifted to states as follows: $S \equiv S'$ holds if $S\, G \equiv S'\, G$ holds for all $G \in Name$.

When a makefile and a state is given in the context, a derived (or dependency) file is one that a derived (or dependency) target of a rule in the makefile maps to under the given state.

## 5 Brute Force Building

Safeness of `make`-based incremental recompilation is *defined* in terms of the reference notion of a brute-force build defined in this section. A brute-force build is the full, unconditional 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

brute-force build, for example when the developer uses targets such as `clobber` of Figure 2 to delete the files that were created in a previous build, and when the advanced user compiles source code before installation.

```
clean:              # R_clean
    rm *.o parser.c
clobber: clean      # R_clobber
    rm pgm
```

**Fig. 2.** Rules $R_{\texttt{clean}}$ and and $R_{\texttt{clobber}}$ may be added to the rules of Figure 1, for cleaning up prior to invoking a brute-force build by deleting intermediate and executable files. Example 3 discusses under what circumstances the commands of the rules are executed.

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

The *make graph* induced by a makefile $M$ is a directed graph where a leaf node is a dependency target which is not derived by a rule in $M$, and a nonleaf node is the target(s) derived by a rule in $M$. There is an edge from node *Ts* to node *Ds* if there is a rule in $M$ that derives a target in *Ts* and depends on a target in *Ds*. A make file is *well-formed* if the induced make graph is acyclic. Figure 3 shows the make graph induced by the makefile of Figure 1.

In the sequel all makefiles are assumed to be well-formed. This is consistent with, e.g., `gnumake` which in the presence of circularity will print a warning message and disregard one of the dependencies.

Rule $R$ is a parent of rule $R'$ if $R'$ derives a target that $R$ depends on. Predecessor/ancestor rules are defined accordingly. Predecessor/ancestor commands are defined in terms of the relationship between the rules that define the commands. We write $M|_T$ for the set of predecessor rules of the rule $R$ deriving target $T$ (including $R$); if $T$ is not derived by a rule in $M$, $M|_T$ is the empty set.

**Definition 1.** *Let $M$ be a makefile. Then command sequence Cs is a* brute-force build *with respect to $M$ of the targets Ds if*

- *command $C$ is in Cs if and only if for some rule $R$ deriving a target $T \in Ds$, $C$ is defined by a rule in $M|_T$.*
- *no command in Cs occurs before any of its predecessor commands.*
- *no command occurs more than once in Cs.*

*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 brute-force build of target `pgm` wrt. to the makefile of Figure 1. So are two permutations of the sequence.

A brute-force build wrt. $M$ is a command sequence which is a brute-force build wrt. $M$ of *some* target list.

**Definition 2 (Safeness).** *File state $S$ is* safe *wrt. makefile $M$ if*

$$exec\, Cs\, S \equiv S$$

*holds for any brute-force build $Cs$ wrt. $M$.*

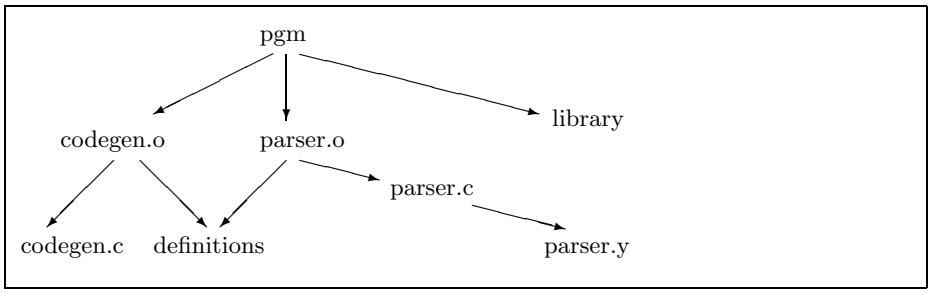


**Fig. 3.** The make graph induced by the makefile of Figure 1.

Thus a file state is safe if the contents of files will remain the same if a brute-force build is invoked. Such a state is what one wants as the outcome of make-based incremental recompilation. In practice, the set of rules relative to which one is interested in safeness is a subset of a given makefile, e.g., the subset $M|_T$ consisting of the rules relevant for building target $T$.

## 6 Satisfiability

Satisfiability of rules is important in the model because a rule fires if and only if it is unsatisfied.

The motivation for using the logical notion of satisfiability is the declarative reading of a makefile as a statement that certain rules must be satisfied. In addition to "rules" and "satisfiability", the informal language commonly used to describe `make` also contains the notion of "derived" targets. Both notions are used in the model because they may help explain `make` execution, not because results from logic are used to infer properties about `make`.

In the literature about `make` – including [4, 16, 12, 6, 18] – there is no definition of `make`'s operational behavior which covers all the special cases. There may be some doubt as to whether a rule fires if, say, its list of dependency targets is empty and its derived file exists.

In order for the `make`-model not to be tied to a specific `make`-variant, the model is parameterized in the satisfiability relation $\models$.

**Definition 3 (Satisfiability).** *A satisfiability relation* $\models$ *is a subset of* $State \times Rule$ *such that for arbitrary states* $S$ *and* $S'$ *we have:*

- *If* $S\ R = S'\ R$*, then*
$$S \models R \Leftrightarrow S' \models R$$
- *If* $R$ *is the rule* $(Gs : Ds; C)$*, where* $Ds \neq nil$*, and* $R'$ *is the rule* $(Gs : Ds'; C')$*, where* $Ds \subseteq Ds'$*, then*
$$S \not\models R \Rightarrow S \not\models R'$$

Satisfiability is lifted to sets of rules: if $M = \{R, R', \ldots\}$, then $S \models M$ means $S \models R$, $S \models R'$, etc.

Thus while real `make` decides satisfiability by comparing time stamps, the `make` model relies only on the above abstract notion of satisfiability, which can be summarized as follows:

- Satisfiability of a rule depends only on the rule's targets.
- Adding targets to a non-empty dependency list preserves unsatisfiability.

For example, the definition of soundness of a rule (see Section 8 below) requires that executing the rule's command renders the rule *satisfied* and its parent rules *unsatisfied* (rather than explicitly requiring a reset of the derived file's time stamp).

While not used inside the model, the following interpretation of satisfiability is assumed in the informal discussion and all examples given in the remainder of the paper. The definition has been reversely engineered from `gnumake`.

**Definition 4 (Interpretation of satisfiability).** *A rule is satisfied in a state if*

- *no dependency file is strictly newer than any derived file, and*
- *all derived files exist, and*
- *all dependency files exist, and*
- *the dependency list is non-empty.*

It may be noted that the above interpretation of satisfiability assumes that $Time$ is linearly ordered (so that time-stamp comparison is well-defined) and that non-existence of files can be expressed. The abstract notion of interpretation is independent of these notions, so we omit their formalization.

*Example 2.* Assume that `codegen.c`, `definitions`, `parser.y`, and `library` exist in $S$, and that none of `codegen.o`, `parser.c`, `parser.o`, and `pgm` exist in $S$ – in other words, only source files and the library exist. Then none of the rules in Figure 1 are satisfied in $S$, because their derived files don't exist.

So-called *phony* targets are targets that never (or practically never) exist as files. The intention is that since the rules that derive them are always unsatisfied, their commands will always be executed.

*Example 3.* See Figure 2. For all $S$ we have $S \not\models R_{\texttt{clean}}$ because the rule has no dependencies. If we assume that `clobber` does not exist in $S$, then we also have that $S \not\models R_{\texttt{clobber}}$. Thus running `make` with target `clobber` will always delete all files except the source files and the external library.

## 7 Semantics of `make`

The semantic definitions model the behavior of `make` in terms of what sequence of rule commands is executed when `make` is invoked.

**Definition 5.** *The semantics of invoking* `make` *with makefile $M$ and initial target $T$ is as follows:*

*Perform a post order traversal of the nonleaf nodes of the induced make graph, starting with the node containing $T$. A visit of node $Gs$ entails the following action: if $Gs$ is derived by rule $R = (Gs : Ds; C) \in M$ and $S \not\models R$ holds, then command $C$ is executed.*

Visiting targets in post order reflects that `make` always processes a rule's dependency targets before testing whether the rule is satisfied.

In addition to the above graph-based definition, a denotational-style definition is given below. By Definition 6 below the value of an expression $\mathbf{M}\,[\![M]\!]\,S\,T$ is the list of commands executed when `make` is invoked with makefile $M$ in the context of state $S$, and with initial target $T$. The denotational definition is more explicit than the graph-based, and it brings out the similarity with logic programs, eg. the semantic definition for constraint logic programs given in [8]. It can be seen from Definition 6 that the main mechanism is the reduction of a list of targets, and there are no references to global variables in the makefile.

More specifically, the format of the definition is partly in the style of denotational semantics [15], including the convention of using $[\![\cdot]\!]$ to distinguish arguments that are syntactical objects. On the other hand, because of the finite nature of `make`'s graph traversal the full machinery of a fixpoint-based definition is not required.

In the definition of $\mathbf{M}$, 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 $\mathbf{T}$ represents the evaluation of a dependency target; note that a makefile contains at most a single rule deriving a given target, so $\mathbf{T}$ is well-defined. The function $\mathbf{R}$ represents rule evaluation.

**Definition 6.** *The semantic function* $\mathbf{M}$ *is defined as follows.*

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

$$
\begin{aligned}
\mathbf{M}\,[\![M]\!]\,S\,T &= \text{let } \langle V, Cs, S'\rangle = \Sigma\,(\mathbf{T}\,[\![M]\!])\,T\,\langle nil, nil, S\rangle \text{ in } Cs \\
\mathbf{T}\,[\![M]\!]\,T\,\langle V, Cs, S\rangle &= \text{if } T \notin V \text{ and } T \in Ts \text{ and } (Ts : Ds; C) \in M \\
&\qquad \text{then } \mathbf{R}\,[\![Ts : Ds; C]\!]\,\langle (V; Ts), Cs, S\rangle \\
&\qquad \text{else } \langle V, Cs, S\rangle \\
\mathbf{R}\,[\![Ts : Ds; C]\!]\,\langle V, Cs, S\rangle &= \text{let } \langle V', Cs', S'\rangle = \Sigma\,(\mathbf{T}\,[\![M]\!])\,Ds\,\langle V, Cs, S\rangle \text{ in} \\
&\qquad \text{if } S' \models (Ts : Ds; C) \text{ then } \langle V', Cs', S'\rangle \\
&\qquad \text{else } \langle V', (Cs'; C), exec\,C\,S'\rangle
\end{aligned}
$$

## 8 Derivability

This section defines three desirable properties of makefile rules, and compounds them in the notion of a build rule. The properties may be summarized as follows:

| Property of a build rule | Expressed as rule of thumb |
|---|---|
| Completeness | The rule's dependency list must contain all the files that the rule's target(s) depend on |
| Fairness | Executing the rule's command may not create or update any targets other than the rule's own |
| Soundness | Executing the rule's command must update the rule's target(s) |

The first and core property of a rule is *completeness* wrt. to a given state. The definition says that the effect on the derived files of executing the rule's command remains the same as long as the content of the dependency files remain the same.

$$complete\ (Gs : Ds; C)\ S \Leftrightarrow \begin{cases} \forall S' : S'\ Ds \equiv S\ Ds \Rightarrow \\ exec\ C\ S'\ Gs \equiv exec\ C\ S\ Gs \end{cases}$$

*Example 4.* Rule $R_{\texttt{pgm}}$ is complete wrt. any state $S$, because its command is

```
cc codegen.o parser.o library -o pgm
```

and by the semantics of linking of `C` object files, definitions of external references are sought only in `parser.o` or `library`, which are both listed as dependency targets. In contrast, rule $R_{\texttt{codegen.o}}$ is complete only in certain states, because its command is

```
cc -c codegen.c
```

and by the semantics of compilation of `C` source files, a preprocessor searches the file `codegen.c` recursively for include directives. Thus the rule, which lists `codegen.c`, and `definitions` as dependencies, is complete only if $S$ is such that no other file than `definitions` is mentioned in an include directive in `codegen.c`, and `definitions` (if it is mentioned) contains no directives to include other files.

Second, *fairness* of a rule wrt. to a state means that executing the rule's command never changes the content or time stamp of any file other than those derived by the rule:

$$fair\ (Gs : Ds; C)\ S \Leftrightarrow \begin{cases} \forall S' : S'\ Ds \equiv S\ Ds \Rightarrow \\ exec\ C\ S'\ (Name \setminus Gs) = S'\ (Name \setminus Gs) \end{cases}$$

*Example 5.* Rules $R_{\texttt{pgm}}$, $R_{\texttt{codegen.o}}$, $R_{\texttt{parser.o}}$, and $R_{\texttt{parser.c}}$ are fair in any state. Rules $R_{\texttt{clean}}$ and $R_{\texttt{clobber}}$ are designed to remove the targets of other rules, and are unfair in any state. For example, firing $R_{\texttt{clobber}}$ removes `pgm`, and even if `pgm` does not exist in the given state $S$ there is a state $S'$ satisfying $S'\texttt{clean} \equiv S\texttt{clean}$ in which that file does exist, so that executing `rm pgm` makes a difference.

Third, *soundness* of a rule wrt. a state means that executing the rule's command renders the rule satisfiable and any parent rule unsatisfiable:

$$sound\ (Gs : Ds; C)\ S \Leftrightarrow \begin{cases} \forall S' : S'\ Ds \equiv S\ Ds \Rightarrow \\ \quad exec\ C\ S' \models (Gs : Ds; C) \\ \quad exec\ C\ S' \not\models R \text{ if } R \text{ is a parent of } (Gs : Ds; C) \end{cases}$$

*Example 6.* $R_{\texttt{codegen.o}}$ is sound wrt. $S$ if the state meets the following two requirements: First, the process of compiling $R_{\texttt{codegen.c}}$ must succeed; then the object file $R_{\texttt{codegen.o}}$ is created or updated, having a time stamp showing that it is newer than the dependency targets $R_{\texttt{codegen.c}}$ and $R_{\texttt{definitions}}$, as well as newer than the target `pgm` derived by the parent rule $R_{\texttt{pgm}}$. Second, the file `definitions` must exist; otherwise $R_{\texttt{codegen.o}}$ is unsatisfied (before and) after executing `cc -c codegen.c`.

**Definition 7 (Build rule).** *Rule $R = (Gs : Ds; C)$ in makefile $M$ is a build rule wrt. $M$ and state $S$, if there is a brute-force build $Cs$ of $Ds$ wrt. $M$ satisfying:*

- *$R$ is complete wrt. $exec\ Cs\ S$*
- *$R$ is fair wrt. $exec\ Cs\ S$*
- *$R$ is sound wrt. $exec\ Cs\ S$*

*Example 7.* If $S$ is as required in Examples 4 and 6, then $R_{\texttt{codegen.o}}$ is sound, fair, and complete wrt. $S$ and $M$. Since none of the rule's dependencies are derived targets, the trivial command *nil* is a brute-force build of the dependency list, so $R_{\texttt{codegen.o}}$ is a build rule wrt. $S$ and $M$.

*Example 8.* To determine whether $R_{\texttt{pgm}}$ is a build rule wrt. $S$ and $M$ it is necessary to examine a state $S'$ obtained from $S$ by executing commands for brute-force building the three dependency targets of $R_{\texttt{pgm}}$. $R_{\texttt{pgm}}$ is fair and complete wrt. any state (see Examples 4 and 5). Thus $R_{\texttt{pgm}}$ is a build rule wrt. $S$ if it is sound wrt. $S'$. The latter holds if $S'$ is such that executing the link command of $R_{\texttt{pgm}}$ does not fail, that is, any external reference of `codegen.o` must be defined in `parser.o` or `library`, with the object files being as created in $S'$.

Derivability ($\vdash$) expresses that all the rules in a makefile are build rules:

**Definition 8 (Derivability ($\vdash$)).** *The targets derived by rules in $M$ are* derivable in *$S$, written $S \vdash M$, if $M$ contains only build rules wrt. $S$ and $M$.*

For the build rule concept to be useful in practice, determination of whether a rule is a build rule should be possible by considering only a single brute-force build of the rule's dependencies, and not the numerous permutations that may exist (see Example 1). For this it suffices that permutations of brute-force builds are equivalent in the sense established by the following lemma:

**Lemma 1.** *Assume $S \vdash M$ and let the permutations $Cs$ and $Cs'$ be brute-force builds wrt. $M$. Then $exec\ Cs'\ S \equiv exec\ Cs\ S$.*

*Proof.* See Appendix B.

This completes the formal framework. The key symbols are listed in the following table.

| Symbol | Definition |
|---|---|
| $\mathcal{Gs} : \mathcal{Ds}; C$ | Rule deriving $\mathcal{Gs}$, depending on $\mathcal{Ds}$, and defining $C$. |
| $S\,G$ | The value of file name $G$ in state $S$. |
| $S\,G \equiv S'\,G$ | The contents of $G$ is the same in $S$ and $S'$. |
| $S\,G = S'\,G$ | The contents and time stamp of $G$ are the same in $S$ and $S'$. |
| $M\|_G$ | The set of rules in $M$ that are predecessors of the rule deriving $G$. |
| $S \models M$ | All rules in $M$ are satisfied in context $S$. |
| $S \vdash M$ | All rules in $M$ are build rules in context $S$. |
| $\mathbf{M}\,[\![M]\!]\,S\,T$ | The command sequence executed by `make` given makefile $M$, state $S$, and target $T$. |

## 9 Safeness

This section contains the main result, Proposition 1 which states sufficient criteria for safeness of `make`-based incremental recompilation. The criteria include that the state against which `make` is invoked is partially safe wrt. the given makefile:

**Definition 9 (Partial safeness).** *File state $S$ is partially safe wrt. $M$ if*

$$\forall\, G \in \mathit{targ}\, M : S \models M|_G \Rightarrow S \textit{ is safe wrt. } M|_G.$$

Clearly, `make` cannot attain safeness when invoked against an arbitrary state, even if the makefile's rules are build rules. For example, if by a mistake a rule's target is "touched" (cf. the Unix `touch` command) upon editing of the dependency files, the state would not qualify as partially safe. Indeed, the rule will not fire, and `make` will not attain safeness. In general, partial safeness shall guarantee that for any derived target $G$, if the rule deriving $G$ does not fire, the state prior to `make` execution must be already safe wrt. the portion of the makefile containing the rule and its predecessors. Note also that a state is trivially safe if it is safe or if all rules are unsatisfied.

*Example 9.* Suppose $S_{safe} \models M$ where $M$ contains the rules of Figures 1 and 2 as in the previous examples. Now assume `parser.y` is edited, yielding state $S$ that satisfies all rules in $M$ except for $R_{\texttt{parser.c}}$. Then $S \not\models M|_{\texttt{parser.c}}$, $S \not\models M|_{\texttt{parser.o}}$, and $S \not\models M|_{\texttt{pgm}}$, while $S \models M|_{\texttt{codegen.o}}$. Thus for $S$ to be partially safe wrt. $M$, we require that $S$ is safe wrt. $M|_{\texttt{codegen.o}}$.

**Proposition 1 (Safeness of make-based incremental recompilation).** *Assume $S \vdash M$ and $S$ is partially safe wrt. $M$. Let $C_{make} = \mathbf{M}\,[\![M]\!]\,S\,T$. Then $\mathit{exec}\, C_{make}\, S$ is safe wrt. $M|_T$.*

*Proof.* See Appendix C.

The remainder of this section shows how partial safeness can be attained (so that Proposition 1 applies) prior to an initial, brute-force `make` invocation as well as prior to subsequent incremental `make` invocations. The difference is only in how partial safeness is attained, while `make` is invoked the same way in all cases.

*The initial brute-force build:*

A sufficient criteria for partial safeness is that no derived files exist. Then all rules are unsatisfied, and so by Proposition 1, if the relevant rules are build rules, the ensuing `make` invocation produces a safe state. To enforce partial safeness prior to an initial brute-force build, makefile rules such as $R_{\texttt{clobber}}$ are sometimes used to delete all derived files.

Additionally, the following lemma shows that the command sequence produced by `make` is indeed a brute-force build if `make` is invoked when all rules are unsatisfied:

**Lemma 2.** *Assume $S \vdash M$ and $S \not\models R$ holds for all $R \in M$. Then the command sequence $\mathbf{M}\,[\![M]\!]\,S\,T$ is a brute-force build of $T$ wrt. $M$.*

*Proof.* See Appendix C.

*Subsequent incremental builds:*

By Proposition 1 the result of the preceding `make` invocation is a state $S_{safe}$ which is safe wrt. the corresponding makefile $M_{safe}$. Since a safe state is (trivially) partially safe, we essentially need to constrain editing so as to preserve partial safeness.

The following proposition gives a sufficient criteria for the new state to be partially safe $S$ wrt. the new makefile $M$. (Recall that $S\,R \equiv S_{safe}\,R$ means that the rule's derived and dependency files have the same contents in $S$ and $S_{safe}$):

**Proposition 2 (Editing constraints).** *Assume $S_{safe}$ is safe wrt. $M_{safe}$, $S_{safe} \vdash M_{safe}$, and $S \vdash M$. Then $S$ is partially safe wrt. $M$ if*

$$\forall\, R \in M : S \models R \Rightarrow \begin{cases} S\,R \equiv S_{safe}\,R \\ M_{safe} \text{ contains a rule defining the same command as } R \end{cases}$$

*Proof.* See Appendix D.

Thus partial safeness is preserved when editing of source files and makefile is constrained as summarized in Table 1. It follows from Proposition 2 that we may safely apply `make`-based incremental recompilation upon such kind of editing.

For source files, Table 1 simply says that editing of a rule's dependency file is permissible if the rule becomes unsatisfied.

With regard to the more subtle question of editing a makefile rule, the table says that one may change the dependency list of a rule without enforcing unsatisfiability of the rule, if the rule's command is not alterned. This applies to removing redundant elements from the dependency list. In addition, it is permissible to add a new rule if the rule is unsatisfied, and to delete a rule.

Note that any number of modifications to source files and makefile may be combined, as long as each modification is permissible on its own.

| Type of editing | Sufficient criteria for partial safeness |
|---|---|
| $S\ D \not\equiv S_{safe}\ D$ (editing of dependency file $D \in Ds$) | $S \not\models R$ |
| $R \notin M_{safe}$ (adding or modifying rule $R$) | $S\ R \equiv S_{safe}\ R$ and $M_{safe}$ contains a rule defining the same command as $R$ |

**Table 1.** A field in the left column indicates a modification which is permissible if the corresponding criterion in the right column is met. Assumptions: $R = (T : Ds; C) \in M$, $S_{safe}$ is safe wrt. $M_{safe}$, $S_{safe} \vdash M_{safe}$, and $S \vdash M$.

*Example 10.* Suppose target `pgm` has been built, so that the state is safe wrt. $M$, and that in subsequent editing a portion of file `codegen.c` is moved to a new file `functions.c`. Accordingly $M$ is modified by adding the rule $R_{\texttt{functions.o}}$ and changing $R_{\texttt{pgm}}$ to $R'_{\texttt{pgm}}$, yielding $M'$ which is equal to $M$ except for the fragments underlined below:

```
functions.o: functions.c definitions                    # R_functions.o
    cc -c functions.c
pgm: codegen.o parser.o library function.o              # R'_pgm
   cc codegen.o parser.o function.o library -o pgm
```

Assume also that all rules are build rules (wrt. the respective states). Then by Proposition 2 the new state is partially safe wrt. $M'$, since $R_{\texttt{codegen.o}}$, $R_{\texttt{functions.o}}$, and $R'_{\texttt{pgm}}$ are unsatisfied upon the editing. It follows from Proposition 1 that invoking `make` to rebuild `pgm` incrementally will produce a safe state. This avoids recompilation of two out of five derived files, even though editing has changed or created two source files, and changed or added two makefile rules.

If the criteria for preserving partial safeness are met, Proposition 1 guarantees that the state produced by incremental recompilation state is again safe, and so repeated cycles are feasible of editing + incremental recompilation.

## 10 Discussion

One may ask whether Definition 8 of build rules is too narrow. For the safeness result to apply to real `make`, the notions of completeness, fairness, and soundness should not (for safety) require build rules to fire too often, and so disqualify makefile rules that are appropriate in practice.

The analysis in Examples 4-8 of Stuart Feldman's example makefile provided indication that the definition is appropriate, because the verification that the rules are build rules made only reasonable assumptions about the contents of the relevant files.

Also, an argument for the validity of the build rule definition is the capture of "cascading" of rule firing from an unsatisfied rule to all its ancestors:

**Lemma 3.** *Let $S \vdash M$, let $T$ be a derived target in $M$, and let $R = (Gs : Ds; C) \in M|_T$. Then*

$$\begin{aligned} & C \in \mathbf{M}\,[\![M]\!]\,S\,T \\ \Leftrightarrow\ & \textit{for some predecessor } R' \textit{ of } R,\ S \not\models R' \textit{ holds.} \end{aligned}$$

*Proof.* See Appendix C.

The above lemma captures "cascading" because $S \not\models M|_G$ holds if and only if $S \not\models R'$ for some predecessor $R'$ of $R$. In particular, the lemma shows that no rules fire except those reached by cascading.

In addition, the following lemma shows that the model captures the fact that `make` creates a state wherein all rules visited are satisfiable (so that none of them will fire if `make` is invoked immediately after).

**Lemma 4.** *Let $S \vdash M$ and let $T$ be a derived target in $M$. Then*

$$\mathit{exec}\,(\mathbf{M}\,[\![M]\!]\,S\,T)\,S \models M|_T$$

*Proof.* See Appendix C.

## 11 Conclusion

The main result is Proposition 1 which states sufficient criteria for `make`-based incremental recompilation to produce a safe state, that is, the same result as a brute-force build. Safeness is shown to hold subject to makefile rules being build rules, and partial safeness of the state against which `make` is invoked. For the use of `make` for incremental recompilation, Proposition 2 provides a constraint on the editing of sources and makefiles as performed upon a previous `make` invocation which ensures that partial safeness, as required by Proposition 1, is attained.

From a practical point of view, the analysis pursued here may be of interest as the basis for guidelines for writing makefiles. The definition of build rules may be translated into rules of thumb for makefile programming, as indicated in Section 8. Examples 4-8 indicate how the definition of build rules can be checked in the case of a makefile for a `C` program.

Also of practical interest are the editing constraints. The permissible modifications include deletion of rules and, under the stronger assumption that rules are rendered unsatisfiable, further changes to rules such as changing their commands. Any combination of these modifications may be performed, as long as each is permissible on its own.

Verification of makefiles is given a strong basis because of the formal approach.

Verification or construction of makefile rules must additionally use knowledge of the semantics of, for example, the commands for `C` compilation. As indicated in Example 4, completeness of a `C` compilation rule cannot be verified simply by checking that all files passed as parameters to the `cc` command are listed

as dependencies. Because of the semantics of the `cc` command, verification or construction of the dependency list also involves parsing of source files, since include directives may establish dependency upon files not passed as parameters.

The definition of the build rule property in terms of a state attained upon execution of the commands of a given rule's predecessor rules (if any) pinpoints a major reason that automated tools for generation of makefile rules may be indispensable in practice. The reason is that a file which is passed as input to, for example, a `cc` command may be created by the command of a predecessor rule, and so is not available for inspection to check for include directives prior to `make` invocation.

The `make` model may support the verification or construction of such rule-generating tools, because the properties that makefile rules should comply with are stated generically in the sense of independently of any particular programming language.

**Acknowledgment.** Thanks to the anonymous referees for many valuable suggestions. The research was supported by the Development Center for Electronic Business and the IT-University, Copenhagen.

## References

1. R. Adams, W. Tichy, and A. Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, Vol. 3 (1), January 1994, 3-28.
2. Demers, A., Reps, T., and Teitelbaum, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. *Proc. Eighth ACM Symposium on Principles of Programming Languages,* Williamsburg, VA, January, 1981, 105-116.
3. M. Elsman. Static Interpretation of Modules. *Proc. International Conference on Functional Programming,* September 99, Paris, France.
4. S. I. Feldman. Make - a program for maintaining computer programs. *Software - Practice and Experience*, Vol. 9, 1979, 255-265.
5. R. Ford and D. Sawamiphakdi. A Greedy Concurrent Approach to Incremental Code Generation. *Proc. 12th Annual ACM Symposium on Principles of Programming Languages,* New Orleans, Louisiana, 1985, 165-178.
6. Institute of Electrical and Electronics Engineers. *Information technology - Portable Operating System Interface (POSIX)* . ANSI/IEEE Std. 1003.2, 1993, Part 2: Shell and Utilities, Volume 1, 1013-1020.
7. N. Jørgensen. *Safeness of Make-Based Incremental Recompilation.* URL: `http://www.ruc.dk/~nielsj/research/papers/make.pdf`.
8. K. Marriott and H. Søndergaard. Analysis of constraint logic programs, *Proc. North American Conference on Logic Programming*, Austin, 1988, 521-540.
9. P. Miller. *Recursive make considered harmful*. URL: `http://www.pcug.org.au/~millerp/rmch/recu-make-cons-harm.html`.
10. The Mozilla build process is described at the URL: `http://www.mozilla.org/tinderbox.html` in the context of a presentation of the build tool "Tinderbox".

11. P.J. Nicklin. Mkmf - makefile editor. *UNIX Programmer's Manual 4.2 BSD*, June 1983.
12. A. Oram and S. Talbott. *Managing projects with make*. O'Reilly, 1993.
13. R. Quinton. *Make and Makefiles*. URL: `http://www.ibiblio.org/pub/docs/unix-tutorials/courses/make.ps`.
14. S. Robbins. *JavaDeps - automatic dependency tracking for Java*. `http://www.cs.mcgill.ca/~stever/software/JavaDeps/`. The JavaDeps tool is a SourceForge project available at `http://sourceforge.net/projects/jmk`.
15. D.A. Schmidt. *Denotational semantics - a methodoogy for language development*. Allyn and Bacon, 1986.
16. R. Stallman and R. McGrath. *GNU Make, Version 3.77*. Free Software Foundation, 1998.
17. W. F. Ticky. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, Vol. 8 (3), July 1986, 273-291.
18. K. Walden. Automatic Generation of Make Dependencies. *Software - Practice and Experience*, Vol. 14 (6), June 1984, 575-585.

URLs available May 10, 2002.

# Appendix

## A Algebra of Commands

Lemma 7-9 establish a command algebra in the sense of criteria that allow for commuting or duplicating commands that are executed on a state.

Recall that by definition of *exec*, the value of $exec\,(C_1; C_2)\,S$ is the same as $exec\,C_2\,(exec\,C_1\,S)$. The first form is preferred because it is more intuitive.

First we note that by definition of rule completeness and soundness we have the following two lemmas:

**Lemma 5 (Simple criteria for rule completeness, fairness, and soundness).** *Suppose $R = (Gs : Ds; C)$ is complete, fair, and sound wrt. $S$, and that $S'$ satisfies $S'\,Ds = S\,Ds$. Then $R$ is complete, fair, and sound wrt. $S'$.*

**Lemma 6 (Simple consequence of completeness and fairness).** *Suppose $R = (Gs : Ds; C)$ is complete and fair wrt. $S$, and that $S' \equiv S$. Then $exec\,C\,S \equiv exec\,C\,S'$.*

**Lemma 7 (Commutativity of unrelated commands).** *Suppose $S \vdash M$ where $M$ contains $R = (Gs : Ds; C)$ and $R_1 = (Gs_1 : Ds_1; C_1)$, and $R$ is complete, fair, and sound wrt. $S$, and $R'$ is complete, fair and sound wrt. $exec\,C\,S$. Also assume that $R_1$ is neither predecessor nor ancestor of $R$. Then*

$$(i)\ \ R \text{ is complete, fair, and sound wrt. } exec\,C_1\,S$$
$$(ii)\ exec\,(C; C_1)\,S \equiv exec\,(C_1; C)\,S$$

*Proof.* (i) holds by Lemma 5 and fairness of $R$ wrt. $S$ using $G \notin Ds_1$.

To establish (ii) we first note that by the same argument given to establish (i) we have that $R_1$ is complete, fair, and sound wrt. $S$. Thus by abuse of notation we say both $R$ and $R_1$ are simply "complete", "fair", and "sound", since $R$ is complete, fair, and sound wrt. $S$ and $exec\,C_1\,S$, and, symmetrically, $R_1$ is wrt. $S$ and $exec\,C\,S$.

By fairness of $R$ and $R_1$, to establish (ii) it suffices to establish

$$exec\,(C; C_1)\,S\,\{G, G_1\} \equiv exec\,(C_1; C)\,S\,\{G, G_1\}$$

To establish the equivalence for $G$, we first note that by fairness of $R_1$ we have

$$exec\,C_1\,S\,Ds \equiv \quad S\,Ds \quad (\text{since } G_1 \notin Ds) \tag{1}$$
$$exec\,(C; C_1)\,S\,G \equiv exec\,C\,S\,G\ (\text{since } G_1 \neq G) \tag{2}$$

By (1) and completeness of $R$ wrt. $S$ we have

$$exec\,(C_1; C)\,S\,G \equiv exec\,C\,S\,G \tag{3}$$

Thus by (3) and (2), using transitivity of $(\equiv)$ we have

$$exec\,(C; C_1)\,S\,G \equiv exec\,(C_1; C)\,S\,G$$

The case of $G_1$ is symmetrical. □

The following lemma is a straightforward generalization of Lemma 7 (proof omitted):

**Lemma 8 (Generalized commutativity of unrelated commands).** *Suppose $S \vdash M$ and $R, R_i \in M$ for $i = 1, \ldots, n$, where $R = (Gs : Ds; C)$ and $R_i = (Gs_i : Ds_i; C_i)$, $R$ is complete, fair, and sound wrt. $S$, and $R_i$ is complete, fair, and sound wrt. $exec\,(C_1 \ldots C_{i-1})\,S$. Then:*

$$\begin{array}{l} \forall j : 1 \leq j \leq n \Rightarrow \\ (i)\ \ R \text{ is complete, fair, and sound wrt. } exec\,(C_1 \ldots C_j)\,S \\ (ii)\ exec\,(C; C_1 \ldots C_n)\,S \equiv exec\,(C_1 \ldots C_j; C; C_{j+1} \ldots C_n)\,S \end{array}$$

**Lemma 9 (Idempotence).** *Consider a rule $R = (Gs : Ds; C)$ which is complete and fair wrt. $S$. Then the equivalence $exec\,(C; C)\,S \equiv exec\,C\,S$ holds.*

*Proof.* Since $M$ is well-formed, so that $G \notin Ds$, we have that the state $exec\,C\,S$ satisfies

$$exec\,C\,S\,Ds = S\,Ds \text{ (by fairness of } R\text{, using } G \notin Ds)$$

Hence the state $exec\,(C; C)\,S$ satisfies the following:

$$\begin{array}{ll} (i) & exec\,(C; C)\,S\,G \equiv exec\,C\,S\,G \quad \text{(by completeness of } R\text{)} \\ (ii) & exec\,(C; C)\,S\,(Name \setminus G) \equiv exec\,C\,S\,(Name \setminus G) \quad \text{(by fairness of } R\text{)} \end{array}$$

□

## B Permutations of Commands in Brute-Force Builds

Lemma 1 of Section 8 stated that the execution of brute-force builds which are permutations produce equivalent file states. The lemma is part (ii) of the following lemma:

**Lemma 10.** *Let $S \vdash M$. Let $Cs = (C_1 \ldots C_n)$ and $Cs'$ be permutations and brute-force builds wrt. $M$. Then*

$$\begin{array}{ll} (i) & \text{for each } R_i = (Gs_i : Ds_i; C_i) \in M : \\ & \quad R_i \text{ is complete, fair, and sound wrt. } exec\,(C_1 \ldots C_{i-1}) \\ (ii) & exec\,Cs'\,S \equiv exec\,Cs\,S \end{array}$$

*Proof.* Let $p(n)$ denote that (i) and (ii) hold if in each of $Cs$ and $Cs'$ the number of commands is at most $n$. We show by induction that $p(n)$ holds for all $n$.
*Base case*: $p(0)$
Trivial because $Cs = Cs' = nil$. (Note that the empty command sequence is a brute-force build of a target derived by a rule whose dependency targets are not derived by other rules, ie. "source" files).
*Inductive case*: $p(n) \Rightarrow p(n+1)$
Assume $p(n)$ and let $Cs$ and $Cs'$ be arbitrary brute-force builds wrt. $M$ containing the same $n+1$ commands. Suppose $C$ is the last command in $Cs'$ and defined

by rule $R = (G : Ds; C)$. Thus $Cs' = Cs''; C$, where $Cs''$ contains $n$ commands. We have that $Cs$ contains $C$, and in addition $n$ commands which we enumerate $C_1 \dots C_n$ based on the order in which they occur in $Cs$. Thus, for some $i$ ($0 \leq i \leq n$) we have

$$Cs = Cs^i \text{ where } Cs^j \text{ is defined by } Cs^j = C_1 \dots C_j; C; C_{j+1} \dots C_n$$

To establish $p(n+1)$ it suffices to show

$$\begin{array}{ll} (i) & R \text{ is complete, fair, and sound wrt. } exec\, Cs''\, S \\ (ii) & exec\, Cs^i\, S \equiv exec\, (Cs''; C)\, S \end{array}$$

Let $i_{min}$ be the smallest $i$ for which $C_1 \dots C_i$ is a brute-force build wrt. $M$ of $Ds$. By Definition 7, since $R$ is a build rule, there is a brute-force builds $Cs_R$ of $Ds$ which is a permutation of $C_1 \dots C_{i_{min}}$ such that $R$ is complete, fair, and sound wrt. $exec\, Cs_R\, S$. Since $i_{min} \leq n$, we have by induction hypothesis that

$$exec\, (C_1 \dots C_{i_{min}})\, S \equiv exec\, Cs_R\, S$$

and so by Lemma 5 we have

$$R \text{ is complete, fair, and sound wrt. } exec\, (C_1 \dots C_{i_{min}})\, S \tag{4}$$

Writing $R_j$ for the rule containing $C_j$, $R$ is a predecessor of no $R_j$ where $j \geq i_{min}$ (since $Cs'$ is a brute-force build), and so $C_1 \dots C_n$ is a brute-force build wrt. $M$, which implies that by induction hypothesis we also have

$$\forall j. 1 \leq j \leq n : R_j \text{ is complete, fair, and sound wrt. } exec\, (C_1 \dots C_{j-1})\, S \tag{5}$$

By (5) and (4) we can apply Lemma 8 (with the "base" state being $exec(C_1 \dots C_{j-1})S$) to infer

$$R \text{ is complete, fair, and sound wrt. } exec\, (C_1 \dots C_n)\, S \tag{6}$$

$$exec\, Cs^i\, S \equiv exec\, (C_1 \dots C_n; C)\, S \tag{7}$$

Since by induction hypothesis ($p(n)$ (ii)) we have

$$exec\, (C_1 \dots C_n)\, S \equiv exec\, Cs''\, S \tag{8}$$

we infer from (6) and Lemma 5 that $R$ is complete, fair, and sound wrt. $execCs''S$ as well. This establishes $p(n+1)$ (i).
Finally, to establish $p(n+1)$ (ii), by (7) it suffices to show

$$exec\, (C_1 \dots C_n; C)\, S \equiv exec\, (Cs''; C)\, S$$

This follows directly from by (6) and (8). □

Lemma 11 is a consequence of Lemma 10, and implies that if $S \vdash M$ and $S$ is safe wrt. $M$, then all rules in $M$ are complete, fair, and sound wrt. $S$.

**Lemma 11 (Rule completeness + fairness + soundness in safe states).** *Suppose $S \vdash M$, and consider a rule $R$ where for each $R'$ deriving a dependency $D$ of $R$, $S$ is safe wrt. $M|_D$. Then $R$ is complete, fair, and sound wrt. $S$.*

## C Safeness: Proposition 1

Proposition 1 as well as Lemma 2-4 rely on Lemma 12-13.

Recall that safeness was defined in Definition 2 in terms of the execution of full brute-force builds. The proof of Proposition 1 is simplified by the following safeness-criterion which allows for considering only the execution of the individual commands in rules:

**Lemma 12 (Safeness criterion).** *Assume $S \vdash M$. Then $S$ is safe wrt. $M$ if and only if*

$$exec\, C\, S \equiv S$$

*holds for any command $C$ occurring in a rule in $M$.*

*Proof.*
(i) $\Rightarrow$ (ii). Consider an arbitrary rule $R = (G : Ds; C)$ in $M$, and let $(Cs; C)$ be a brute-force build of $G$. By (i) we have

$$S \equiv exec\, Cs\, S \tag{9}$$

$$S \equiv exec\, (Cs; C)\, S \tag{10}$$

By Lemma 10 we have that $R$ is complete wrt. $S$, so

$$\begin{aligned} exec\, C\, S &\equiv exec\, (Cs; C)\, S && \text{(by (9) and Lemma 6)} \\ &\equiv S && \text{(by (10))} \end{aligned}$$

(ii) $\Rightarrow$ (i).
Assume $S \vdash M$ and (ii). Let $p(n)$ denote the proposition that the equivalence

$$exec\, Cs\, S \equiv S$$

holds if the number of commands in the brute-force build $Cs$ is at most $n$. It suffices to show that $p(n)$ holds for all $n$, which we show by induction over $n$:
*Base case:* $p(0)$
In this case $Cs$ is the empty command sequence $nil$, so the equivalence holds trivially.
*Inductive case:* $p(n) \Rightarrow p(n+1)$
Consider an arbitrary brute-force build $Cs$ containing $n+1$ commands. Let $C$ be the last command in the sequence, so that $Cs = (Cs'; C)$ where $Cs'$ contains $n$ commands. Let the rule containing $C$ be $R = (G : Ds; C)$. We must show

$$exec\, (Cs'; C)\, S \equiv S \tag{11}$$

By induction hypothesis we have

$$exec\, Cs'\, S \equiv S \tag{12}$$

By Lemma 10 we have that $R$ is complete wrt. $exec Cs'\, S$, so by (12) and Lemma 6 we have

$$exec\, (Cs'; C)\ \equiv exec\, C\, S \tag{13}$$

By (ii) we have

$$exec\, C\, S \equiv S \tag{14}$$

Thus we can infer (11) from (13) and (14) using transitivity of ($\equiv$). □

The following lemma is the basis for Proposition 1 and Lemma 2-4.

**Lemma 13.** *Assume $S \vdash M$ and $S$ is partially safe wrt. $M$. Let $N$ be the number of nonleaf nodes in the make graph of $M|_T$, and consider a post order traversal of those $N$ nodes starting with the node containing $T$. For $0 \leq n \leq N$, let $S_n$ be the state attained upon visiting the first $n$ nodes (with $S_0 = S$), and let $R_n = (Gs_n : Ds_n; C_n)$ be the rule deriving the targets of the $n$'th node. Let $C_{make} = \mathbf{M}\ [\![M]\!]\ S\ T$, and let $p(n)$ denote the conjunction of the following propositions:*

$$\begin{array}{ll}
p_a(n) : S \not\models M|_{R_n} & \Leftrightarrow C_n \in C_{make} \\
p_b(n) : \forall i : i \leq n & \Rightarrow S_n \models R_i \\
p_c(n) : \forall i : i \leq n \wedge S \models R_i & \Rightarrow S_n\, Gs_i = S\, Gs_i \\
p_d(n) : \forall i, j : i < n \wedge (j \leq i \vee j > n) & \Rightarrow S_n\, Gs_j = S_i\, Gs_j \\
p_e(n) : \forall i : i \leq n & \Rightarrow exec\, C_i\, S_n \equiv S_n
\end{array}$$

*Then $p(n)$ holds for all $n$ where $1 \leq n \leq N$.*

Specifically, Proposition 1 is $p_e(N)$, Lemma 2 follows from $p_a(N)$, Lemma 3 is $p_a(N)$, and Lemma 4 is $p_b(N)$.

Lemma 13 says that during traversal of the induced make graph, upon visiting the $n$'th target $G_n$ the following holds invariantly for $1 \leq n \leq N$:

$p_a(n)$ $R_n$ fired if and only if $R_n$ has a predecessor rule not satisfied in $S$.
$p_b(n)$ For any node visited so far, the corresponding rule is satisfied in the current state.
$p_c(n)$ For any node visited so far, if the corresponding rule was satisfied in $S$, its derived files remains unaltered.
$p_d(n)$ In the current state and the state attained upon the visit to the $i$'th node, all derived files are the same, except for derived files of rules corresponding to a node visited after the $i$'th visit.
$p_e(n)$ For any rule $R$ corresponding to a node visited so far, the current state is safe wrt. $M|_R$.

The proof of Lemma 13 is by induction over $n$. The idea is to consider two distinct cases: First, in the case of $S \models M|_{R_n}$ use that $exec\, C_n\, S \equiv S$ holds by partial safeness, and second, in the case of $S \not\models M|_{R_n}$ use that $R_n$ fires.

*Proof. Base case*: $p(1)$
$p_a(1)$: Since $R_1$ is the first rule evaluated, it is evaluated against the initial state $S$, and so by assumption $S \vdash M$ $R_1$ is complete, fair, and sound wrt. $S$.

For $p_a(1)$ - $p_e(1)$ we consider two distinct cases:

(i) $S \models M|_{R_1}$:
$p_a(1)$: Since $R_1$ is evaluated against $S$, it does not fire by assumption (i).
$p_b(1)$-$p_d(1)$: Not firing $R_1$ implies $S_1 = S$, which establishes all three invariants.
$p_e(1)$: Since $S_1 = S$, the equivalence $p_e(1)$ becomes

$$exec\, C_1\, S \equiv S$$

which holds by partial safeness of $S$ and assumption $S \models M|_{R_1}$.
(ii) $S \not\models M|_{R_1}$:
$p_a(1)$: Since $R_1$ is evaluated against $S$, the rule fires by assumption (ii).
$p_b(1)$: holds by soundness of $R_1$ wrt. $S$.
$p_c(1)$: holds trivially by assumption (ii).
$p_d(1)$: holds by fairness of $R_1$ wrt. $S$.
$p_e(1)$: The equivalence $p_e(1)$ becomes

$$exec\, (C_1; C_1)\, S \equiv exec\, C_1\, S$$

Since $R_1$ is fair and complete wrt. $S$, this equivalence holds by Lemma 9.

*Inductive case*: $(\forall i \leq n : p(i)) \Rightarrow p(n+1)$

We first note that $R_{n+1}$ is evaluated against state $S_n$ which by $p_e(n)$ is safe wrt. $M|_D$ for any $D \in Ds_{n+1}$. Thus by Lemma 11, $R_{n+1}$ is complete, fair, and sound wrt. $S_n$.

For $p_a(1)$ - $p_e(1)$ we consider, as above, two distinct cases:
(i) $S \models M|_{R_{n+1}}$:
$p_a(n+1)$: By (i) we have $S \models R_{n+1}$. Recall that $R = (G_{n+1} : Ds_{n+1}; C_{n+1})$, where $j \leq n$ for each $D_j \in Ds_{n+1}$. Thus

$$\begin{aligned} S_n\, Ds_{n+1} &= S\, Ds_{n+1} \text{ (by } p_c(n)) \\ S_n\, G_{n+1} &= S\, G_{n+1} \text{ (by } p_d(n)) \end{aligned}$$

Thus by Definition 3 of satisfiability we have $S_n \models R_{n+1}$. Therefore $R_{n+1}$ does not fire.
$p_b(n+1)$: Since $R_{n+1}$ does not fire, we have $S_{n+1} = S_n$. Thus $\forall i \leq n : S_{n+1} \models R_i$ holds by $p_b(n)$, and $S_{n+1} \models R_{n+1}$ holds by $S_n \models R_{n+1}$.
$p_c(n+1)$: We must establish $S \models R_i \Rightarrow S_{n+1}\, G_i = S\, G_i$ for $i \leq n+1$. The implication holds for $i \leq n$ by $p_c(n)$ and using $S_{n+1} = S_n$. Since by assumption (i) we have $S \models R_{n+1}$, it remains to establish:

$$\begin{aligned} S_{n+1}\, G_{n+1} &= S_n\, G_{n+1} \text{ (since } S_{n+1} = S_n) \\ &= S\, G_{n+1} \text{ (by } p_d(n)) \end{aligned}$$

$p_d(n+1)$: Follows directly from $p_d(n)$ and $S_{n+1} = S_n$.
$p_e(n+1)$: We must establish

$$\forall i \leq n+1 : exec\, C_i\, S_{n+1} \equiv S_{n+1}$$

Since $S_{n+1} = S_n$, the equivalence holds for $i \leq n$ by $p_e(n)$. For $i = n+1$ we have, with $G \neq G_{n+1}$:

$$\begin{array}{lll} exec\, C_{n+1}\, S_n\, G & & \\ \quad = S_n\, G & \text{(by fairness of } R_{n+1} \text{ wrt. } S_n) & \\ exec\, C_{n+1}\, S_n\, G_{n+1} & & \\ \quad \equiv exec\, C_{n+1}\, S\, G_{n+1} & \text{(by compl. of } R_{n+1} \text{ wrt. } S_n, \text{ using } S_n\, Ds_n = S\, Ds_n) & \\ \quad \equiv S\, G_{n+1} & \text{(by partial safeness of } S \text{ wrt. } M, \text{ using } S \models M|_{R_{n+1}}) & \\ \quad \equiv S_n\, G_{n+1} & \text{(by } p_d(n)) & \end{array}$$

(ii) $S \not\models M|_{R_{n+1}}$:

$p_a(n+1)$: We consider two distinct sub-cases:

First, assume, in addition to (ii), that $S \models M|_{R_j}$ holds for any rule $R_j$ deriving a dependency target of $R_{n+1}$. Thus for this subcase of (i) we can infer, as in (ii), that $S_n\, Ds_{n+1} = S\, Ds_{n+1}$ and $S_n\, G_{n+1} = S\, G_{n+1}$. Since in this subcase we have $S \not\models R_{n+1}$, it follows that $S_n \not\models R_{n+1}$, so $R_{n+1}$ fires.

Second, assume $S \not\models M|_{R_j}$ for some $R_j$ deriving a dependency target of $R_{n+1}$. By $p_a(j)$ we have $C_j \in C_{make}$, implying that $R_j$ fired because $S_{j-1} \not\models R_j$. By $p_a(j)$, $R_j$ was sound wrt. $S_{j-1}$, so firing of $R_j$ implies $S_j \not\models R_{n+1}$. By $p_d(n)$ we have $S_n\, \{G_j, G_{n+1}\} = S_j\, \{G_j, G_{n+1}\}$, which by definition of rule soundness implies $S_n \not\models R_{n+1}$, so $R_{n+1}$ fires. Thus $p_a(n+1)$ holds.

$p_b(n+1)$: Since $R_{n+1}$ fires, $S_{n+1} \models R_{n+1}$ holds by soundness of $R_{n+1}$ wrt. $S_n$. By $p_b(n)$ and fairness of $R_{n+1}$ wrt. $S_n$, $S_{n+1} \models R_i$ holds for any $i \leq n$.
$p_c(n+1)$: This holds trivially by assumption (ii).
$p_d(n+1)$: Follows from $p_d(n)$ and fairness of $R_{n+1}$ wrt. $S_n$.
$p_e(n+1)$: We must show that for $i \leq n+1$

$$exec\, C_i\, S_{n+1} \equiv S_{n+1}$$

For $i = n+1$ we have

$$\begin{array}{lll} exec\, C_{n+1}\, S_{n+1} \equiv exec\, (C_{n+1}; C_{n+1})\, S_n & \text{(since } R_{n+1} \text{ fires)} & \\ \quad \equiv exec\, C_{n+1}\, S_n & \text{(by Lemma 9 and } p_a(n+1)) & \\ \quad \equiv exec\, S_{n+1} & \text{(since } R_{n+1} \text{ fires)} & \end{array}$$

For arbitrary $i \leq n$ we consider rule $R_i = (G_i : Ds_i; C_i)$. Since $j \leq i-1$ holds for each $D_j \in Ds_i$ we have:

$$S_{n+1}\, Ds_i = S_{i-1}\, Ds_i \tag{15}$$

$$S_n\, Ds_i = S_{i-1}\, Ds_i \tag{16}$$

and so for $G \neq G_i$

$$\begin{array}{lll} exec\, C_i\, S_{n+1}\, G \equiv S_{n+1}\, G & \text{(by (15) and fairness of } R_i \text{ wrt. } S_n) & \\ exec\, C_i\, S_{n+1}\, G_i \equiv exec\, C_i\, S_{i-1}\, G_i & \text{(by (15) and completeness of } R_i \text{ wrt. } S_{i-1}) & \\ \quad \equiv exec\, C_i\, S_n\, G_i & \text{(by (16) and completeness of } R_i \text{ wrt. } S_{i-1}) & \\ \quad \equiv S_n\, G_i & \text{(by } p_d(n)) & \end{array}$$

□

## D Editing Constraints: Proposition 2

This section proves Proposition 2 which says that $S$ is partially safe wrt. $M$ if

$$(*)\ \begin{cases} S_{safe} \text{ is safe wrt. } M_{safe} \\ S_{safe} \vdash M_{safe} \\ S \vdash M \\ \forall\, R \in M : S \models R \Rightarrow \begin{cases} S\ R \equiv S_{safe}\ R \\ \exists R' \in M_{safe} : R \text{ and } R' \text{ define the same command} \end{cases} \end{cases}$$

*Proof.* Assume $(*)$. By Definition 9, it suffices to show that for an arbitrary derived target $T$ in $M$, where $S \models M|_T$ holds, we have that $S$ is safe wrt. $M|_T$. By Lemma 12 this holds if

$$exec\ C\ S \equiv S \tag{17}$$

where $C$ is the command occurring in an arbitrary rule $R = (Gs : Ds; C) \in M|_T$.

The proof idea is to establish and use that

$$exec\ C\ S_{safe} \equiv S_{safe} \tag{18}$$

From assumption $S \models M|_T$ and $R \in M|_T$ we infer $S \models R$. Thus by $(*)$ there is a rule $R' \in M_{safe}$ defining the same command $C$, and by safeness of $S_{safe}$ wrt. $M_{safe}$ the equivalence (18) follows from Lemma 12.

Moreover, by $S\ R \equiv S_{safe}\ R$ we have

$$S\ Ds \equiv S_{safe}\ Ds \tag{19}$$
$$S\ Gs \equiv S_{safe}\ Gs \tag{20}$$

Since $R$ is fair and complete wrt. $S_{safe}$ (by Lemma 11) we have

$$\begin{array}{lll} exec\ C\ S\ (Name \setminus Gs) & \equiv S\ (Name \setminus Gs) & \text{(by (19) and fairness of } R \text{ wrt. } S_{safe}) \\ exec\ C\ S\ Gs & \equiv exec\ C\ S_{safe}\ Gs & \text{(by (19) and completeness of } R \text{ wrt. } S_{safe}) \\ & \equiv S_{safe}\ Gs & \text{(by (18))} \\ & \equiv S\ Gs & \text{(by (20))} \end{array}$$

which establishes (17). □