Abstracting Dependencies between Software Configuration Items

CARL A. GUNTER
University of Pennsylvania

This article studies an abstract model of dependencies between software configuration items based on a theory of concurrent computation over a class of Petri nets called *production* nets. A general theory of build optimizations and their correctness is developed based on a form of abstract interpretation called a *build abstraction*; these are created during a build and are used to optimize subsequent builds. Various examples of such optimizations are discussed. The theory is used to show how correctness properties can be characterized and proved, and how optimizations can be composed and compared.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Languages, Management, Verification

Additional Key Words and Phrases: Abstract interpretation, mathematical models of build dependencies, Petri nets, software configuration management

1. INTRODUCTION

Even a modest software project entails the creation of a collection of what are sometimes called *software configuration items*. Such items may be held in files, one item per file, or they may be more abstractly described and stored. A characteristic example is the collection of source, object, executable binary, and archive files that arise in a programming project. Some of the files are directly modified by a programmer, while others are produced by the use of tools such as a compiler or other processing tool. Certain of these produced items are ones the project ultimately ships as the deliver-

The following agencies, company, and institute provided partial support for this project: ARO (USA), EPSRC (UK), NIMS (UK), NSF (USA), Oki Electric Industry Co., Ltd. (Japan), ONR (USA).
Author's address: School of Engineering and Applied Science, Department of Computer and Information Science, University of Pennsylvania, 200 South 33rd Street, Philadelphia, PA 19104–6389; email: gunter@cis.upenn.edu.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2000 ACM 1049-331X/00/0100–0094 $5.00

```
table : a.out indata
     a.out

a.out : main.o datanal.o lo.o /usr/cg208/lib/gen.a
     cc main.o datanal.o lo.o /usr/cg208/lib/gen.a

main.o : main.c
     cc -c main.c

datanal.o : datanal.c
     cc -c datanal.c

lo.o : lo.s
     as -o lo.o lo.s
```

Fig. 1. Sample Makefile.

able of the effort. It is essential, therefore, that the implications of any changes in the source items be properly reflected in the items directly or indirectly produced from them. This can become an overwhelming task if the project environment does not provide automated support for it. A recognition of the ubiquity of this problem, and the insight that a tool could address it in a wide range of cases, led Stuart Feldman to develop the Unix tool, `make` [Feldman 1978]. In this limited application domain, the special-purpose `make` *description files* were easier to write and maintain than general-purpose programs, so the tool quickly gained widespread use as a *software configuration management* (SCM) tool. An example of a `make` description file for a small configuration of C programming files appears in Figure 1. The lines with the colons express the dependencies between the software configuration items of interest.[1] The lines indented by tabs indicate how the *target*, the file to the left of the colon, is to be produced from its *prerequisites*, the files to the right of the colon. The `make` description file, thus, records the dependencies between software configuration items and the actions required to establish consistency of the system based on these dependencies.

To be practical, it is essential for the configuration management system to optimize system builds by recognizing circumstances where the rebuilding of a component is unnecessary. The `make` tool uses a simple and effective strategy for this based on the most recent modification dates of the prerequisites for a target. Roughly, if they are older than that of the target, then it is viewed that the target does not need to be rebuilt from its prerequisites. Since most projects proceed by a sequence of comparatively

[1]It is convenient not to identify software configuration items as a general concept with operating system files. However, `make` essentially does make this identification, with various advantages and disadvantages.

small modifications interleaved with builds of parts of the system, this optimization saves a great deal of needless evaluation. Moreover, it is very *general*, since it does not require `make` to "understand" anything about the items themselves or the operations being performed on them. Let us refer to optimizations with this property as *semantics-independent*.

Since `make` was introduced, it has been extended many times and has seen substantial competition from its chief rival approach, the *Integrated Development Environment* (IDE) (see Fowler [1990] for an analysis of the chief challenges to `make`). IDEs are often capable of maintaining dependencies automatically through an "understanding" of the semantics of the systems they integrate. An IDE may maintain its own `make`-like description file, which is not directly modified by the programmer, and derive this file automatically using information about the semantics of the underlying items. For instance, it may be possible to automatically derive information about dependencies between a family of C files based on facts about C bindings. The advantage of such a system is that the programmer is relieved of the tedius and error-prone manual maintenance of the description file. However, a disadvantage is that an IDE may not integrate all of the component production tools a project requires. For instance, software coded in C may be accompanied by files that contain documentation in SGML, which has a quite different semantics from C and hence a rather different way in which dependencies must be derived. This balance extends also to the area of build optimizations, where the semantics of the underlying items may be very helpful in avoiding builds. In a seminal work on this idea, Tichy [1986] noted that it is often possible to avoid builds on Pascal modules when it can be seen that a modification of a target's prerequisites was actually "irrelevant" to that target. For instance, the prerequisite might be a library where the modification is the addition of a new procedure used only in other places in the software. Of course, this requires that knowledge of the semantics of Pascal be represented in the build tool. Let us refer to such optimizations as *semantics-dependent*.

A key problem with the variety of build optimizations is that it is difficult to understand their correctness properties individually and almost impossible to understand them when the optimizations are being *combined*. Tichy's system, for instance, uses a mixture of the `make` date optimization and the Pascal-specific optimization. Although there is a sketch of a correctness proof in Tichy [1986], it is not detailed enough to count as a mathematical proof. More recently there has been progress on treatments of build optimizations that are mathematically rigorous and, indeed, able to support proofs of *completeness*. For instance, it was quickly recognized that Tichy's analysis in Tichy [1986] could be strengthened [Schwanke and Kaiser 1988], but Shao and Appel [1993] were able to claim that their algorithm produced an optimal point in this space for the SML programming language. On the other hand, Abadi et al. [1996] are able to prove a completeness theorem for a class of semantics-independent caching optimizations for the lambda-calculus, on which the Vesta [Hanna and Levin 1993] SCM tool is based. However, neither of these treatments fully

explores the implications of Tichy [1986], which provides a modular combination of semantics-dependent and semantics-independent optimizations.

This article introduces a new approach to modeling builds and a broad class of build optimizations. There are two basic ideas. The first is to use a modeling technique from concurrent and distributed systems called a *Petri net* to model dependencies between software configuration items. A concurrency formalism has been chosen because the typical state of a build configuration contains a great deal of potential coarse-grained parallelism, so it is natural to model the build directly as a concurrent computation. Also this system is simple and similar to what is currently used in most SCM systems. The second basic idea is to view build abstractions as being based on the *abstract interpretation* of configuration items. Each item of the system will have one standard interpretation (as a C program, an SGML document, or whatever) and possibly many abstract interpretations (a date, a fingerprint, information about functions that are used, and so on). The progress of a build is to consult the abstract interpretations to see if any of them can show that a particular build of a target from prerequisites is unnecessary. If this fails then the build occurs, and both the standard and abstract interpretations of the targets are all updated. The article formalizes this architecture, derives correctness properties and basic theorems to aid proofs of correctness, and shows how to apply the technique to a variety of examples. In particular, a theory of how to combine abstractions is described, and it is shown that the general framework can be used to prove theorems that compare different build optimizations.

The article has the following goals:

(1) To develop a theory of build optimizations general enough so as to encompass both semantics-dependent and semantics-independent optimizations.

(2) To show how this theory can be used to compare and combine build optimizations.

(3) To provide ideas about how extensible build optimization mechanisms can be supported by SCM tools and safely applied by the users of such tools.

SCM tools must deal with a host of issues such as the organization of versions and variants [Cottam 1984; DuraSoft 1993; Perry 1987; Rochkind 1975; Tichy 1985] and providing "process" support [ANSI/IEEE 1987; IEEE 1990], in addition to the topic discussed in this article, which is support for building targets from their prerequisites. Thus, additional work will be required to determine how to integrate these functions with extensible build optimizations. An adequate treatment of the latter concept is the proper place to begin.

The structure of the article is as follows. After this introduction, the second section describes a structure called a *production net (p-net)* which is used to describe dependencies and uses this formalism to discuss a variety of pragmantic build-optimization ideas that have been considered over the

years as the result of configuration maintenance experience. The third section provides formal definitions of p-net models and abstractions. The fourth section considers the integration and comparison of different models and abstractions. The final section surveys some of the other work on SCM models and tools in comparison to the current work.

2. PRODUCTION NETS

In this section the graph structures that are used to represent relationships between software configuration items are introduced. Their models are described informally and illustrated with examples.

We start with the following concept:

Definition (Nets). A *net* is a four-tuple $N = (B, E, S, T)$ where

—B is a finite set of *variables,*

—E is a finite set of *events,*

—$S \subseteq B \times E$ is the *prerequisite* relation, and

—$T \subseteq E \times B$ is the *target* relation.

This differs slightly from the usual Petri net terminology, where variables are called *conditions* because they represent the conditions under which events that take them as prerequisites can occur. The term "variables" is more evocative for the application in this article, however. The letters u, v, x, y, z and the like are used to denote both variables in B and events in E, whereas letters e, f, g are used for events in E. For a net $N = (B, E, S, T)$, it is convenient to write $x\ S\ e$ for $(x, e) \in S$ and to write $e\ T\ x$ for $(e, x) \in T$. When working with more than one net, subscripts and/or superscripts can be used to distinguish parts of the respective nets, e.g., $N' = (B', E', S', T')$. However, when it is clear which net is meant for the various sets and relations, the following notation is more succinct. Given a net $N = (B, E, S, T)$, and an event $e \in E$, we define ${}^\bullet e = \{x \mid x\ S\ e\}$ and $e^\bullet = \{x \mid e\ T\ x\}$. These are respectively called the *prerequisites* and *targets* of event e. Figure 2 shows an example of a net using the Petri net figure conventions: circles are used for variables; rectangles are used for events; relations in S are arrows pointing into a rectangle; and relations in T are arrows pointing into a circle. In the Figure 2 example, ${}^\bullet e = \{x, u\}$ while $e^\bullet = \{y, z\}$. It is also convenient to have a notation for the union of the prerequisites and targets of an event, so we define ${}^\bullet e^\bullet = {}^\bullet e \cup e^\bullet$.

Intuitively a software configuration is modeled as a net by treating software items—like `main.o` and `main.c` in the description file in Figure 1—as net variables. On the other hand, operations—such as the application of the compiler `cc` with the switch `-c` to the input `main.c`—are treated as events. A fairly readable informal notation that works well with ASCII characters is to write the names of items (variables) in parentheses, which

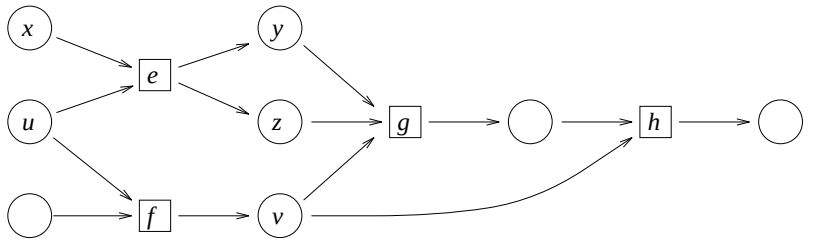

Fig. 2. Example of a net.

are reminiscent of circles, and the names of operations (events) in brackets, which are reminiscent of rectangles. The `make` description file of Figure 1 is depicted by the net in Figure 3.

To provide a tractable theory for our purposes here, nets must satisfy certain axioms. The directed graph defined by a net N has $B \cup E$ as its nodes and $S \cup T$ as its edges. A *cycle* in a directed graph is a sequence of nodes $x_0, \ldots, x_n$ such that there is an edge from x_n to x_0 and such that, for each $i < n$, there is an edge from x_i to x_{i+1}. A directed graph is said to be *acyclic* if it has no cycles. A net is said to be acyclic if the directed graph it defines is acyclic.

Definition (P-Nets). A net $N = (B, E, S, T)$ is a *production net (p-net)* if it has the following properties:

(1) It is acyclic.

(2) (Unique Producer.) If $e\ T\ x$ and $e'\ T\ x$, then $e = e'$.

(3) For each $e \in E$, both ${}^\bullet e$ and $e^\bullet$ are nonempty.

(4) For each $x \in B$, there is some e such that $x \in {}^\bullet e$ or $x \in e^\bullet$.

The nets in Figures 2 and 3 are both production nets. Figure 4 gives several examples of the ways in which a net can fail to be a p-net. In Figure 4, net (a) contains a cycle; net (b) fails to satisfy the unique-producer condition (2); nets (c), (d), and (e) have an event with prerequisites or targets that are unacceptable for condition (2); and net (f) has an isolated variable, in violation of condition (4).

Borison [1987] defines graphs underlying what she calls "configurations" that are the same as p-nets except for the nontriviality conditions ((3) and (4)). Her configuration graphs are acyclic nets that satisfy the unique-producer condition and where each event has a target.

The unique-producer condition (2) is equivalent to saying that the set $T^{op} = \{(x, e) \mid e\ T\ x\}$ is a partial function. The set of elements on which this partial function is defined are those variables x such that there is a (unique) event e such that $e\ T\ x$; in this case we say that x is a target variable and that e is the *event that produced* x. The set of elements on which T^{op} is *not* defined are of particular importance because, intuitively,

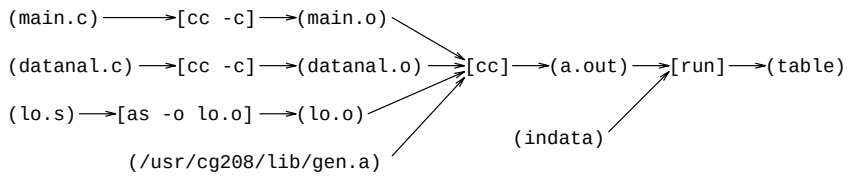

Fig. 3. Net corresponding to Makefile in Figure 1.

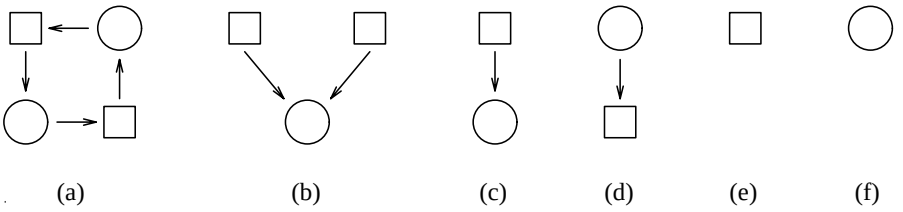

Fig. 4. Nets that are not production nets.

they are the ones that are modified by the environment (*viz.* programmers). Given a variable x, if there is no event e such that $e\ T\ x$, then x is said to be a *source* variable. In Figure 2, x and u are source variables, while y, z, and v are target variables. Variables y and z are produced by event e, while variable v is produced by event f.

With the formalism of p-nets it is possible to clarify some of the notions of "dependence graph" that one finds (usually without definition) in the SCM literature. There are three kinds of dependence graph of interest, based on whether one is interested in variables only, events only, or variables and events together. Since the graph determined by a production net $N = (B, E, S, T)$ is acyclic, the transitive and reflexive closure of its edge relation defines a poset $\sqsubseteq_N$ relation on $B \cup E$. The restrictions of $\sqsubseteq_N$ to variables and events are respectively denoted $\sqsubseteq_B$ and $\sqsubseteq_E$. For example, in Figure 2, we have $x \sqsubseteq_N e \sqsubseteq_N y \sqsubseteq_N g$ whereas variables y, z are incomparable (with respect to $\sqsubseteq_N$ and $\sqsubseteq_B$) and events e, f are incomparable (with respect to $\sqsubseteq_N$ and $\sqsubseteq_E$). The `make` description file uses the colon notation to define the relation $\sqsubseteq_B$ on variables. However, the evaluation of a `make` file is based on events (actions) structured by the dependencies in $\sqsubseteq_E$.

For any p-net N, a subset $X \subseteq B \cup E$ is *left-closed* (relative to $\sqsubseteq_N$) if $x \in X$ and $y \sqsubseteq_N x$ implies $y \in X$. The notion of state for computation on a production net N will be modeled by left-closed subsets of $\sqsubseteq_N$ that satisfy one additional condition.

Definition (Markings). Let $N = (B, E, S, T)$ be a production net. A subset $M \subseteq B \cup E$ is *target-closed* if, for every event $e \in E$, $e^\bullet \cap M \neq \emptyset$ implies $e^\bullet \subseteq M$. A *marking* M for N is a subset of $B \cup E$ that is both left-closed (with respect to $\sqsubseteq_N$) and target-closed.

An event is viewed as having one of three states relative to a marking.

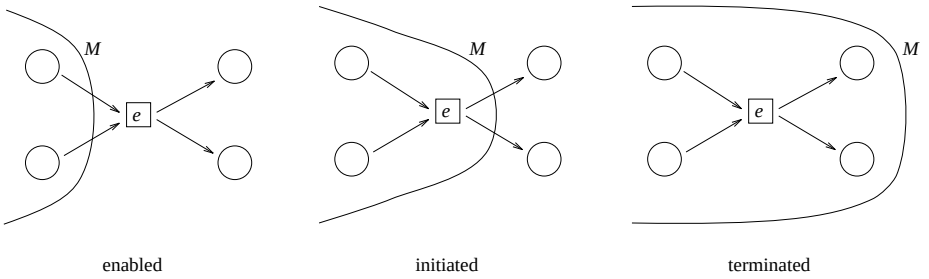

Fig. 5. Three relationships between marking M and event e.

Definition (Event States). Let M be a marking on a p-net $N = (B, E, S, T)$, and let e be an event. We say that e is *enabled* by M and write M/e if ${}^\bullet e \subseteq M$ and $e \notin M$. We say that e is *initiated* in M and write e/M if $e \in M$ and $e^\bullet \cap M = \emptyset$. We say that e is *terminated* in M and write $e \searrow M$ if $e^\bullet \subseteq M$.

Note that e is terminated in M if, and only if, $M \cap e^\bullet \neq \emptyset$. The intuition is that M/e is a state in which the prerequisites of e have been built, but where the event e has not yet begun. The state e/M is one in which e has begun, but has not yet finished. The state $e \searrow M$ is one in which e has finished and now its targets are within M. A pictorial representation of the different states appears in Figure 5. The desired closure property is expressed as follows.

LEMMA 1. *Suppose M is a marking of a p-net $N = (B, E, S, T)$. The following three implications hold:*

(1) *If M/e, then $M' = M \cup \{e\}$ is a marking with e/M'.*

(2) *If e/M, then $M' = M \cup e^\bullet$ is a marking with $e \searrow M'$.*

(3) *If M/e, then $M' = M \cup \{e\} \cup e^\bullet$ is a marking with $e \searrow M'$.*

A proof of the lemma can be found in the Appendix.

Let us now turn to how computation over p-nets will be represented. The idea is similar to providing a set of operational rules for computation over a Petri net but specializing the rules for the application at hand. Consider the net in Figure 2 for example: note the way u is shared by e and f and the way v is shared by g and h. As a build computes a result, the prerequisite remains intact through the remainder of the build; sharing in the sense of these examples does not introduce the Petri net concept of "conflict," that is, competition of events for prerequisites. So, the usual Petri net semantics of placing markings on variables and having them consumed by events to which they are prerequisites is not a convenient way of thinking of system state for builds. Instead, one wants to view a build as having achieved consistency between prerequisites and targets in a subset of the p-net that is left-closed relative to $\sqsubseteq_B$.

In this section the semantics of p-nets will be described informally based on examples. In subsequent sections a more formal treatment which will be applied to some of the examples. The main concept is that of the state of a p-net, and how a computation over the p-net changes the state. The state is simply an assignment of values to the variables of the p-net. There is a distinguished state for the "standard" model, and a collection of additional states representing other assignments of values to the variables for purposes of optimizing the build. These additional assignments are called abstractions and will be changed as the computation progresses. The basic ideas can all be illustrated by explaining how a build of the components in Figure 1 is modeled. Let us refer to the net in question as N for the purposes of this discussion. B consists of 10 variables, and E of 5 events. The standard state of the p-net is defined by values associated with its variables, which, in this case, are the contents of certain files. For instance, the value of `main.c` is the contents of the file `main.c`. The abstract state of the p-net, using the `make` date abstraction, is the assignment of modification dates to these files. A slight complexity arises from the fact that both of these assignments may be partial, because, for example, `main.o` may not yet have been created by the compiler.

Let us now look at how a computation occurs over the p-net of Figure 1. Assume at first that none of the targets have been created and that the standard state consists only of an assignment of values to the source files (including `data`). The abstract state is an assignment of dates to these variables. A build of `table` procedes by the invocation of any event whose prerequisites exist; for instance `main.c` could be used to produce `main.o`. When this occurs the abstract value of the variable `main.o` is assigned a date that is more recent than that of `main.c`. At this stage, the "built" part of the p-net consists of the source files together with the variable `main.o` and the event that produced it. Now, let us assume that the build of `table` was completed and that two operations occur to change the state of the p-net. First, the file `main.c` is changed; in this case the abstract value (the date) of the variable is also changed to reflect the more recent modification date, which is now more recent than the date associated with `main.o`. Second, the file `lo.o` is deleted; this also results in the deletion of its abstract value. If a build is now requested of `table`, then it procedes as it did before in producing `lo.o`, where the event is triggered after it is observed that the target does not exist. In the case of `main.o`, however, the target is found to exist, so its abstract value is checked to determine if the build actually needs to occur. When it is found that this value is earlier than that of `main.c`, the build is carried out, and the date of `main.o` is changed. In this state of affairs the target is said to be "out-of-date," which means that the date optimization fails to find that the rebuild is unnecessary. By contrast, when the abstract value of `datanal.o` is examined, it is found to have the desired relationship to the abstract values of its prerequisites—it is "up-to-date"—so the event to produce it is not evaluated; and both the standard and abstract values of `datanal.o` are left unchanged. At

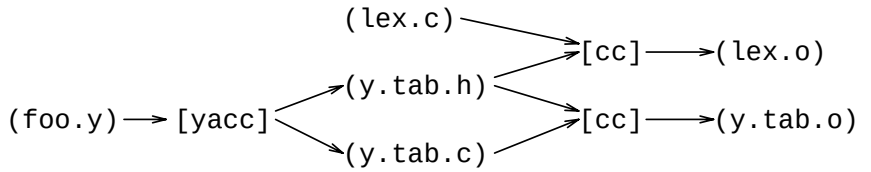

Fig. 6. Identity of old and new inputs.

this point, the marking of the p-net is taken to include `datanal.o` and the event that produces it, even though the event was not evaluated. This is the optimization function of the date abstraction.

In anticipation of the mathematical treatment of this story, note that we require mathematical definitions of state (both standard and abstract) and of the build computation. The former will be carried out with the mathematical concept of a section, which is essentially just an assignment of values to variables with the restriction that this assignment must respect the types of the variables. The latter will be done by means of a labeled reduction system, a common means of modeling state transitions on concurrent systems, including Petri nets. None of these details is essential to understand the examples below.

The build process above is a familiar one, but merits examination about what is essential and what is incidental. It is essential that there exist operations to update both the standard and abstract values of the variables. It is also essential that the property of the abstract values used to avoid a rebuild must preserve the integrity of the targets. That is, if this property says that a rebuild is unnecessary, then this must really be the case. On the other hand, it is not essential that the rebuilds *only* occur when necessary: false negatives are an allowed result of the abstraction optimization, although the fewer of them encountered the better. In particular, it is incidental that the abstract values are dates: they could be anything that satisfies the essential properties. We now turn to some examples to show the limitations of dates as an abstraction optimization, and consider some of the other possibilities. The first example concerns parsing tools; the second concerns SML programming language compilations; the third concerns C header files; the fourth concerns rebuilding from archives; and the final example concerns rebuilding the theories of an automated deduction tool.

Figure 6 illustrates a production net which arises in instances where one is using the parser-generator `yacc`. The `yacc` tool takes an input file in a special format and produces a C source file, `y.tab.c`, and a C header file `y.tab.h`. The header file describes the information about keywords declared in `foo.y`, which is all of the information generated from this file that is needed to create the lexer, `lex.o`. The input file `foo.y` also describes a possibly intricate collection of actions used to determine the parsing in `y.tab.o`. This collection of actions often requires debugging or optimization, so the actions in `foo.y` may be modified much more frequently than the keyword declarations there. However, if only the actions

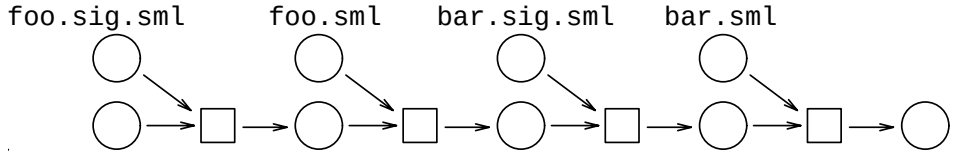

Fig. 7. Hash keys to avoid cascading recompilation.

in `foo.y` have been modified, then the produced header file `y.tab.h` will not change. Nevertheless, its *date* will change with the new generation, thus making the lexer object file out-of-date and thereby inducing a recompilation of `lex.c` when the lexer object file is again required. The effort is wasted though, because the file `y.tab.h` did not really change, only its date did.

Another example, this one involving the SML programming language, appears in Figure 7. Unlike typical C programs, SML programs generally have deeply nested dependencies, often dozens of files long. In the figure, the SML signature `FOO` in the file `foo.sig.sml`, together with some basic environment, is compiled into a target environment. The implementing structure `Foo` of this signature is then compiled and incorporated into this environment. After this, a signature `BAR`, which is stored in a file `bar-.sig.sml` and uses names from `FOO` and `Foo`, is compiled and incorporated. Finally, its implementing structure `Bar`, which is in a file `bar.sml` and uses names from `FOO`, `Foo`, and `BAR`, is compiled and incorporated. Now, if even a *comment* is changed in `foo.sig.sml`, then all of the steps used to produce this final target will need to be rerun if one uses only the standard `make` date optimization. When dealing with such deeply nested dependencies, it becomes worthwhile to retain information sufficient to recognize when it is probable that the cascading sequence of recompilations can be cut off. In the Compilation Manager (CM) IDE of Blume [1998], which is part of the SML/NJ system, targets of compilations are assigned "fingerprints." A fingerprint is a bit sequence computed from a file in such a way that files with the same fingerprint are very unlikely to be different.[2] This provides a pragmatic aid when development is under way; even the low probability of error due to the imperfection of the fingerprint assignment can be eliminated by deleting the target files to induce complete regeneration before final testing. Time saved avoiding recompilations quickly repays the overhead of calculating the fingerprints in typical SML programming projects.

A somewhat more subtle issue of dependence is illustrated in Figure 8. Header files allow separate compilation of C programs. Given a collection of C files $C_1, \ldots, C_n$ and a collection of corresponding header files $H_1, \ldots, H_n$, one seeks to organize things so that any given file C_i can be compiled with a suitable subset of the header files. In particular, no other C

[2]The method for calculating fingerprints used in CM is based on Rabin's CRC polynomials; see Broder [1993] for an exposition.

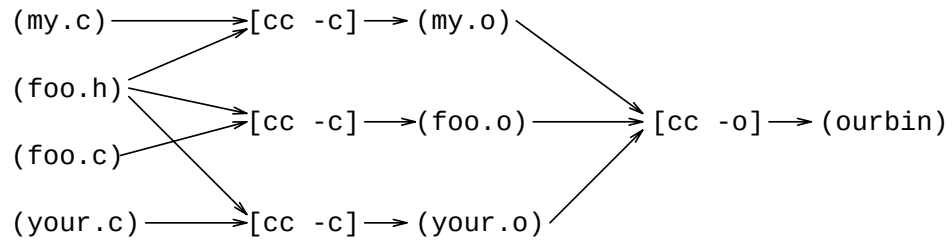

Fig. 8. Changes in headers.

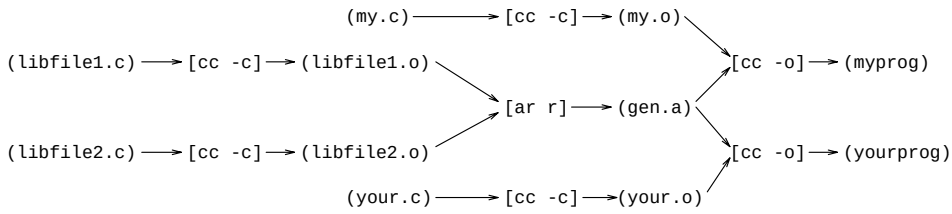

Fig. 9. Coarse library dependencies.

files are required. This has the advantage that one may compile C_i even if the C implementations of header files are not available, perhaps because they are under development. In Figure 8 compilation of `my.c` and `your.c` can be done using `foo.h`. If `foo.c` is modified then the files `my.o` and `your.o` do not become out-of-date as a result, although the linking of the three object files does become out-of-date. This provides a valuable form of support for separate compilation, allowing significant independence between programmers as well as an opportunity for the use of parallelism in builds. Suppose, however, that the programmer in charge of `your.c` asks that an additional function `f` be included in `foo.c`, and its prototype placed in `foo.h`. This results in a change in `foo.h`, causing the compilation of `my.c` to become out-of-date. However, the program `my.c` may not make any use of `f` or depend on it in any way. An intuitive and inexpensive approach to recognizing this state of affairs automatically was introduced by Tichy [1986] under the sobriquet "smart" recompilation. Tichy's benchmarks suggest that the analysis is well worth the time spent on it, as one might intuitively predict. His implementation was actually for Pascal with modules rather than C programs, so the example of Figure 8 should be taken with an appropriate grain of salt, but the basic idea applies to most programming languages. The idea has inspired several subsequent studies, including one on "smart*er*" recompilation [Schwanke and Kaiser 1988] for C programs and "smart*est*" recompilation for SML [Shao and Appel 1993].

Another common C programming issue concerns dependencies on library archives. Consider the p-net in Figure 9. Library archives such as the file `gen.a` in the figure were introduced as a means to cope with the proliferation of object files (and the space they occupy). Unfortunately, when a library is under development, the use of the library by clients results in unexpected compilation dependencies. Looking at Figure 9, let us assume

that `my.c` uses functions from `libfile1.c` but not from `libfile2.c`, whereas `your.c` uses functions from `libfile2.c` but not from `libfile1.c`. In this case, a change in `libfile2.c` is as likely to be recompiled by a request for target `myprog` as it is by a request for target `yourprog`, but only the latter really needs the recompilation to be done. This problem has proved severe enough that some versions of `make` provide a special notation for reporting this sort of finer library dependency in the `make` description file [Oram and Talbott 1991]. For instance, one can write `gen.a(libfile1.c)` as the prerequisite for `myprog` in order to indicate which part of the library archive `myprog` depends on. In this case, recompilation only occurs when `libfile1.c` is out-of-date and not when other parts of the library, on which `myprog` does not depend, are out-of-date. This provides a way to circumvent the usual `make` identification of files and software configuration items.

Configuration management issues are not particular to compiling source code: let us conclude with an example of a somewhat different flavor. Many systems include configuration items that are basically data rather than programs but where the data has its own complex set of dependencies and must be processed in ways that resemble compilation. For example, theorem-proving systems generally rely on a base of mathematical facts organized into modules. A case in point is the HOL90 interactive theorem-proving system, whose configuration maintenance problems are discussed in Gunter [1988]. In addition to its underlying program source code, which implements user interfaces, decision procedures, parsing, and so on, the HOL90 system includes a body of formalized mathematics organized into software configuration items called *theories*. Theories are related to one another in a graph structure that indicates one or more parent theories for each theory except for a root theory called `hol`. Among the things a theory defines are *theorems*, which are mathematical facts that have been proved using HOL90, and *constant definitions*. It is typical for theories to grow over time as new theorems and constant definitions are added. When a theory is changed in this way, none of its descendants needs to be rebuilt (that is, checked for truth). However, when the definition of a constant is changed, then all descendent theories that use the constant must be rebuilt. The latter change is far less common than the former: it is typical for a change in a parent theory to have no affect on its descendants. Thus an abstraction that caches information about a change to take advantage of this common-case monotonicity property would substantially reduce the need for rebuilding.

3. P-NET MODELS AND ABSTRACTIONS

To really represent the kinds of issues described in the previous section, it is essential to provide a *model* for a production net that describes the kinds of entities involved in a build over the net and the relations they are expected to satisfy. By way of illustration, consider the very basic p-net shown in Figure 10. This net has many models. For example, it might be

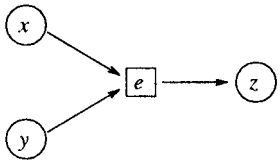

Fig. 10. P-net example.

the case that x is interpreted as a C source file, y as a header file, and z as an object file. The event e is interpreted as the relation between the input files and output files which holds when the interpretation of the output could be the outcome of a correct C compilation of the interpretations of its inputs. In another model, x is interpreted as an object file, y as a file containing data, and z as another data file. The desired relation is that z (that is, the interpretation of z) should be the result of using x to process the data in y. This view allows one to understand more precisely the role that the labels were meant to play in something like Figure 3: they suggest the intended model of the underlying p-net. One can get along with leaving the model as an informal concept up to a certain point, but a more precise interpretation requires more structure. In particular, the goal of this section is to explore "abstract interpretations" of dependencies between software configuration items. As in other cases, such as the well-known application of strictness analysis [Burn et al. 1986], an abstract interpretation is based on the use of a "nonstandard" model which, to be useful, is simpler in certain regards than the "standard" model but retains key relationships to the standard model. Regarding the example above, the value of x may be a C file, but its abstract interpretation may be its *modification date*. This is the abstract interpretation exploited by `make`.

To provide the key definitions, some mathematical machinery is required. An *indexed family of sets* is an indexing collection I together with a function associating with each element $i \in I$ a set S_i. Such an indexed family will be written $S = (S_i \mid i \in I)$, and we say that S is "indexed over I." A *section* $s = (s_i \mid i \in I)$ of such an indexed family of sets S is a function associating with each $i \in I$ an element $s_i \in S_i$. The *product* $\Pi(S_i \mid i \in I)$ is the set of all sections of S. A *partial* section of an indexed family of sets $(S_i \mid i \in I)$ is a section $s = (s_i \mid i \in I')$ of $(S_i \mid i \in I')$ where $I' \subseteq I$. In this case I' is called the *domain of existence* of s, and we say that s_i *exists* if $i \in I'$. The *partial product* $\tilde{\Pi}(S_i \mid i \in I)$ is the set of all partial sections of S. We will generally be concerned with partial sections.

A model of a p-net is a family of sets indexed over a subset of the variables of the net and a family of relations indexed over a subset of the events of the net.

Definition (Models). A *model* $\mathcal{A} = (B', E', V, R)$ of a p-net $N = (B, E, S, T)$ is

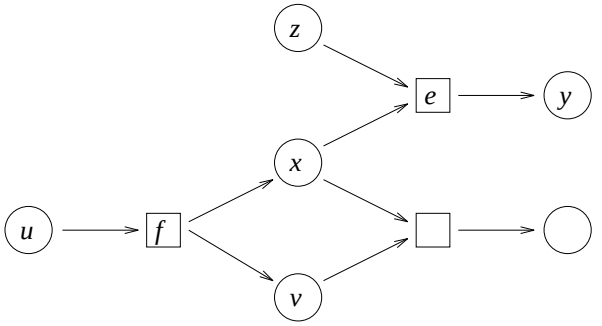

Fig. 11. P-net for example of a model.

—a family of sets $V = (V_x \mid x \in B')$ indexed by variables $B' \subseteq B$ and

—a family of relations $R = (R_e \mid e \in E')$ indexed by events $E' \subseteq E$

such that, for each $e \in E'$, we have ${}^\bullet e^\bullet \subseteq B'$ and

$$R_e \subseteq \tilde{\Pi}(V_x \mid x \in {}^\bullet e) \times \tilde{\Pi}(V_x \mid x \in e^\bullet).$$

$\mathcal{A}$ is a *total* model of N if $B = B'$ and $E = E'$.

When dealing with multiple models, variables, events, etc., can be distinguished by superscripts: $\mathcal{A} = (B^{\mathcal{A}}, E^{\mathcal{A}}, V^{\mathcal{A}}, R^{\mathcal{A}})$.

To clarify ideas, let us consider an example of a model. Consider the production net in Figure 11, which corresponds to the one in Figure 6. The "standard" model for the variables u, v, x, y, z associates sets as follows: V_u is the set of `yacc` input files; V_x is the set of C header files; both V_v and V_z are the set of C files; and V_y is the set of object files. The events e, f are interpreted as follows: R_f is the relation between `yacc` input files and output files; R_e is the relation between C input files and the object files produced by compiling them.[3]

Given a model, the association of variables and events to elements of the model provides the concept of system state. Some order-theoretic notation is useful in the definition. Given a subset $X \subseteq B \cup E$, for a given p-net N, the *left-closure* of X is the set

$$\downarrow X = \{y \in B \cup E \mid \exists x \in X.\ \sqsubseteq_N x\}.$$

Definition 6 (States and Consistent Events). Suppose

$$\mathcal{A} = (B', E', V, R)$$

[3]A technical quibble here is that each of the sets V_i in this example is really the set of all possible file contents. In particular, any sort of file could be input to `yacc` by a programmer having a bad day, so the view that there is a special set of `yacc` inputs is slightly misleading. The case is less misleading for target files, which must be in the specific class of files that could be produced by programs like the C compiler.

is a model of the p-net $N = (B, E, S, T)$. A *state* of $\mathcal{A}$ is a partial section of $(V_x \mid x \in B')$. Given a state s of $\mathcal{A}$ we write $\mathcal{A}, s \vDash e$ if, and only if,

$$(s_i \mid i \in {}^\bullet e)\ R_e\ (s_i \mid i \in e^\bullet).$$

An event e is said to be *consistent* in state s if $\mathcal{A}, s \vDash e$. A left-closed subset L of $B \cup E$ is consistent with respect to s if, for each target variable x in L, the event that produced x is consistent. A marking on N is said to be consistent if its left-closure relative to $\sqsubseteq_N$ is consistent.

To give some help on the notation here, we can think of $\mathcal{A}$ as being a multisorted model of first-order logic that interprets relation symbols R_e corresponding to events $e \in E'$. Variables corresponding to variables are interpreted in $\mathcal{A}$ by the state s. For example, for the net in Figure 2, $\mathcal{A}, s \vDash e$ could be viewed as shorthand for $\mathcal{A}, s \vDash R_e(u, x, y, z)$. If the relation R_e is date abstraction then this relation holds if $s(x)$ and $s(u)$ are earlier than $s(y)$ and $s(z)$. Note, however, that the state s can be partial on the variables, and the variables have different types (sorts); the relation may still hold even if s is undefined on one of the variables x, u, y, z. Unlike first-order models, the relations in the p-net model do not provide any order for the variables in the way they appear in a relation expression like $R_e(x, u, y, z)$.

An important property of the relation $\mathcal{A}, s \vDash e$ is that it is dependent only on the values in ${}^\bullet e^\bullet$. It is helpful to introduce some notation. Given a partial function $f : X \to Y$ and a subset U of the domain of f, we will denote by $f \mid U$ the restriction of f to U. It usually does not matter whether $f \mid U$ is to be viewed as a partial function with U as its domain or whether the domain of $f \mid U$ is X, while f is undefined outside of U. For the purposes of this article, however, it is taken to be the latter. The proof of the following can be obtained by simply unrolling the definition:

LEMMA 2. *Let $\mathcal{A}$ be a model of a p-net N, and let e be an event in N. For any pair of states s, s', if $\mathcal{A}, s' \vDash e$ and $(s \mid {}^\bullet e^\bullet) = (s' \mid {}^\bullet e^\bullet)$, then $\mathcal{A}, s \vDash e$.*

Our theory of system builds will involve two kinds of mathematical entities. First, there are the *relations* or *invariants* which embody the correctness property of the build and its optimizations. Second, there are the *servers* which drive the computation. We begin with the definition of the kinds of invariants required. The goal is to describe a relation between a pair of models $\mathcal{A}$ and $\mathcal{B}$ for a given net N, wherein $\mathcal{B}$ can be viewed as an *abstraction* of $\mathcal{A}$. The motivating example is the `make` date abstraction: the model $\mathcal{A}$ is the "standard" model in which variables are interpreted as things like C source files and object files, while the model $\mathcal{B}$ instead interprets these variables as *dates*. The relations on $\mathcal{A}$ are things like "input source file x compiles to output object file y," while the relations on

$\mathcal{B}$ are things like "the date of x is earlier than the date of y." More precisely, what we need is a relation between states s of $\mathcal{A}$ and states t of $\mathcal{B}$ such that the relation holds when t is to be viewed as a correct abstraction of s. Here is the precise formulation:

Definition. Suppose $\mathcal{A} = (B_1, E_1, S, T)$ and $\mathcal{B} = (B_2, E_2, S, T)$ are models of the p-net $N = (B, E, S, T)$ such that $B_2 \subseteq B_1$ and $E_2 \subseteq E_1$. An *abstraction* $\Phi : \mathcal{A} \to \mathcal{B}$ is a relation between $\mathcal{A}$-states and $\mathcal{B}$-states that satisfies the following rules for each $\mathcal{A}$-state s, $\mathcal{B}$-state t, and subset $U \subseteq B$:

$$[\text{Production}] \quad \frac{\Phi(s, t) \qquad \mathcal{B}, t \vDash e}{A, s \vDash e}$$

$$[\text{Deletion}] \quad \frac{\Phi(s, t)}{\Phi(s \mid U, t \mid U)}$$

The first of these two rules is soundness *with respect to production*, and the second is soundness *with respect to deletion*.

To understand the names and origins of the two rules, we must appreciate the invariants expected of a system for which an abstraction will be used. First of all, the aim of an abstraction is to signal when a production does *not* need to be performed. To be sound, it must be the case that if $\mathcal{B}$ says that a production step is not needed, then the corresponding $\mathcal{A}$ values are consistent. Hence, soundness "with respect to production" is the basic correctness criterion. To understand the second rule, consider the invariants that `make` is expected to satisfy. Source files may be modified (or possibly even deleted), while target files may be deleted *but not modified*. Modifying a target file would generally be an unusual thing to do and might very well result in an incorrect build. Refering to the description file in Figure 1, suppose, for example, that after modifying `main.c`, one "touched" the file `main.o`, thus giving it a more recent date than `main.c`. A build using the date optimization would then fail to update the target object file to consistency with the source file on which it depends: an incorrect build would result. On the other hand, *deleting* the object file would not cause a problem, because the deleted file would be properly rebuilt from the up-to-date source file. It is common to delete target files to save space for example. Source files may also be deleted, since it will sometimes be the case that an event does not require one or more of its inputs to be defined. Thus soundness "with respect to deletion" is a natural requirement to impose on abstractions.

To carry out a system build it is essential to have a collection of *servers* that can produce the desired outputs from the available inputs. For instance, the description file in Figure 1 requires a C compiler to process C source files and an assembler to process assembly code. When one is dealing with abstractions, another server is needed to calculate them. In the case of the `make` date optimization, this task consists only of noting the

date. However, some of the other examples discussed in the previous section require a more sophisticated collection of operations. For example, smart recompilation [Tichy 1986] requires a "history" attribute which is used to cache information required for assessing the effect of a change. In each of the ways a value in a state may change, a server is required to recalculate an abstraction that takes the change into account. There are three cases for a change: a source variable is modified by the environment; a target variable is produced in the course of a build; or a variable is deleted. In the last case the corresponding abstraction values are deleted (made undefined), and correctness is ensured by the rule for soundness with respect to deletion; so no special server is required. The former pair of cases applies to a set U of source items, or to a set $e^\bullet$ of target items. We need a name for this pair of cases:

Definition. Let $N = (B, E, S, T)$ be a p-net. A *change set* of variables is a subset $U \subseteq B$ such that

(1) each element of U is a source *or*

(2) there is an event e such that $U = e^\bullet$.

We may now define the key server concepts. First, some notation is needed. It is useful for us to consider the restriction of f to the *complement* of the set U. This is denoted $f \mid U$.

Definition. Suppose $\mathcal{A}$ and $\mathcal{B}$ are models of the p-net $N = (B, E, S, T)$, and suppose $\Phi : \mathcal{A} \to \mathcal{B}$ is an abstraction.

A *build server for* $\mathcal{A}$ is a function α which takes as its arguments an event e and $\mathcal{A}$-state s and returns as its value an $\mathcal{A}$-state s' such that $s \curlywedge e^\bullet = s' \curlywedge e^\bullet$ and $\mathcal{A}, s' \vDash e$.

An *abstraction server for* Φ is a function ϕ which takes as its arguments a change set of variables, an $\mathcal{A}$-state, and a $\mathcal{B}$-state; it returns as its value a $\mathcal{B}$-state. Abstractions must satisfy the following rule for any pair of $\mathcal{A}$-states s, s', change set $U \subseteq B$, and $\mathcal{B}$-state t:

$$[\text{Abstraction}] \quad \frac{s \curlywedge U = s' \curlywedge U \qquad \downarrow U \text{ is consistent in } s' \qquad \Phi(s, t)}{\Phi(s', \phi(U, s', t))}$$

With these definitions it is possible to describe the operational rules for p-net computation in Figure 12. *Initiation* occurs when a build must be carried out because the abstraction $\mathcal{B}$ does *not* indicate that it can be omitted. When *termination* occurs, the build result and the abstraction are updated by α and ϕ respectively in the new state. *Omission* occurs when the abstraction $\mathcal{B}$ indicates that a rebuild is unnecessary.

Note that an abstraction has an operational semantics only if it has a server, so it will sometimes be convenient to write $\Phi, \phi : \mathcal{A} \to \mathcal{B}$ to indicate that ϕ is a server for abstraction Φ. Not much about the existence

$$\text{[Initiation]} \quad \frac{M/e \qquad \mathcal{B}, t \not\models e}{M, s, t \xrightarrow{\hat{e}} M \cup \{e\}, s, t}$$

$$\text{[Termination]} \quad \frac{e/M \qquad s' = \alpha(e, s)}{M, s, t \xrightarrow{\check{e}} M \cup e^{\bullet}, s', \phi(e^{\bullet}, s', t)}$$

$$\text{[Omission]} \quad \frac{M/e \qquad \mathcal{B}, t \models e}{M, s, t \xrightarrow{\epsilon} M \cup \{e\} \cup e^{\bullet}, s, t}$$

Fig. 12. Computation relative to servers α and ϕ.

of a *useful* ϕ needs to be said here, since an operator that simply causes the abstract values to be undefined would suffice as a server in many cases. Later, we will introduce the concept of "progressive" server as one which makes a certain form of "progress" as the build proceeds. Some of the servers we consider below are progressive, and others are not. However, the principle soundness result for the rules in Figure 12 applies to all relations and servers. Let $\rightarrow^*$ denote the transitive closure of the evaluation relation $\rightarrow$.

THEOREM 3. *Let* $\Phi : \mathcal{A} \rightarrow \mathcal{B}$ *be an abstraction between models of a p-net* $N = (B, E, S, T)$. *Suppose* s *is a state of* $\mathcal{A}$, *and suppose* t *is a state of* $\mathcal{B}$ *such that* $\Phi(s, t)$. *Let* M *be a consistent marking of* $\mathcal{A}$, $\mathcal{S}$. *If* $M, s, t \rightarrow^* M', s', t'$ *with respect to a build server* α *for* $\mathcal{A}$ *and an abstraction server* ϕ *for* Φ, *then*

(1) M' *is a consistent marking of* $\mathcal{A}$, s' *and*

(2) $\Phi(s', t')$.

The proof of the theorem appears in the Appendix.

It remains to understand the relationship between builds on a configuration and this soundness result. A configuration over a p-net begins life with a state in which only a collection of source items are defined. At this point the abstraction server can be invoked on this change set of source items to create an abstraction that satisfies the hypotheses of Theorem 3. A build based on the operational semantics in Figure 12 can then be initiated starting with this standard and abstract state. After this, the standard configuration state may be altered by the change of standard source items and the deletion of any standard items. Before a rebuild, the abstract state is changed by the application of the abstraction server to any altered source items and the restriction of the abstraction server to the domain of existence of the standard state. These changes preserve the invariants because of the rules for the abstraction server and the deletion rule for abstractions. Since the hypotheses of Theorem 3 are again satisfied, the subsequent rebuild preserves the invariant.

4. APPLICATIONS OF ABSTRACTIONS

An application of the concept of an abstraction requires the demonstration of a model, an abstraction relation, and an abstraction server. Suppose we are given a model $\mathcal{A}$ for a p-net $N = (B, E, S, T)$. To define an abstraction for $\mathcal{A}$ we need the following:

—*Abstraction Model:* We must define a model $\mathcal{B}$ for N which is to serve as the space of abstractions. This entails selecting the variables $B^{\mathcal{B}}$ that are to be abstracted, and the events $E^{\mathcal{B}}$ for which the abstractions are to be tested. For each element $x \in B^{\mathcal{B}}$, a space $V_x^{\mathcal{B}}$ of abstract values is required, and for each $e \in E^{\mathcal{B}}$, a relation $R_e^{\mathcal{B}}$ between $\mathcal{B}$-states of the prerequisites and targets of e is required. It is fair game to use $V_x^{\mathcal{A}}$ to define $V_x^{\mathcal{B}}$, although it will be unusual to use $R_e^{\mathcal{A}}$ to define $R_e^{\mathcal{B}}$. The relationship between $R_e^{\mathcal{A}}$ and $R_e^{\mathcal{B}}$ is most likely to be expressed in the abstraction relation.

—*Abstraction Relation:* We must define the relation Φ between $\mathcal{A}$-states and $\mathcal{B}$-states. This relation needs to satisfy the two rules for abstraction relations, but it may also involve other properties that are to be assumed as invariants preserved by the abstraction server.

—*Abstraction Server:* It is necessary to define a server function ϕ for the abstraction Φ. If the abstraction Φ is chosen unwisely, it may be difficult or impossible to find a feasibly computable server for it.

Having selected these three things, it still remains a question of the application itself whether the abstraction will be *useful*. The axioms for abstraction relations and servers ensure only that the abstraction optimization is *sound*.

4.1 Date Abstraction

The date abstraction was explained informally before. To render it precisely, we use the set ω of nonnegative numbers $(0, 1, 2, 3, \ldots)$ to represent dates. An event is up-to-date if the maximum of its prerequisites is less than the minimum of its targets. The abstraction server must change the dates after any rebuild to ensure that the new date is the most recent one in both the set of variables that are *below* the updated variable in the dependency order $\sqsubseteq_B$ and those that are *above* it in that order. Let $\mathcal{A}$ be a total model for a p-net $N = (B, E, S, T)$.

—*Model:* The model $\mathcal{B}$ takes $B^{\mathcal{B}} = B$ and $E^{\mathcal{B}} = E$. For each $x \in B$, we define $V_x^{\mathcal{B}} = \omega$. For each event $e \in E$, define $(t \mid {}^\bullet e)\ R_e^{\mathcal{B}}\ (t \mid e^\bullet)$ to hold if, and only if, t is defined on ${}^\bullet e^\bullet$ and

$$\max\{t_x \mid x \in {}^\bullet e\} < \min\{t_y \mid y \in e^\bullet\}. \tag{1}$$

—*Relation:* The abstraction relation is defined by stipulating that $\Phi(s, t)$ holds if, and only if, for every event e, such that t is defined on ${}^\bullet e^\bullet$ and Inequality (1) holds, the $\mathcal{A}$-state s is also defined on ${}^\bullet e^\bullet$ and $\mathcal{A}, s \models e$.

—*Server:* Suppose U is a change set of variables, and suppose s' is an $\mathcal{A}$-state, while t is a $\mathcal{B}$-state. The state $t' = \phi(U, s', t)$ has the same values as t outside of U. For each $x \in U$, if s'_x is defined, then

$$t'_x = 1 + \max\{t_x \mid x \in \downarrow\{x\} \cup \uparrow\{x\}\} \tag{2}$$

where $\uparrow\{x\}$ means the set of variables y such that $x \sqsubseteq_B y$. That is, the date on x is "later" than anything related to it by $\sqsubseteq_B$. If s'_x is undefined, then t'_x is also taken to be undefined.

Our proof burdens are to show that Φ is an abstraction relation and that ϕ is an abstraction server for Φ. In effect, this means proving that the rules [Production], [Deletion], and [Abstraction] are satisfied by $\mathcal{B}$, Φ, and ϕ. Let us do this fully for this example. We start with soundness with respect to production:

$$\frac{\Phi(s, t) \quad \mathcal{B}, t \models e}{\mathcal{A}, s \models e}$$

This rule has essentially been defined to hold for this example.[4] If $\mathcal{B}, t \models e$ then t is defined on ${}^\bullet e^\bullet$, and Inequality (1) holds. By the definition of $\Phi(s, t)$ these conditions imply $\mathcal{A}, s \models \mathcal{C}$. To see that it is sound with respect to deletion, suppose $U \subseteq B$; we must show that the following rule is satisfied:

$$\frac{\Phi(s, t)}{\Phi(s \mid U, t \mid U)}$$

Suppose the hypothesis of the rule holds; let $s' = s \mid U$ and $t' = t \mid U$, and suppose $e \in E$. If t' is defined on ${}^\bullet e^\bullet$ then it must be the case that ${}^\bullet e^\bullet \subseteq U$ so t is also defined on ${}^\bullet e^\bullet$, where it has the same value as t'. If s' is undefined on any of the elements of ${}^\bullet e^\bullet$, then so is s, hence also t (because $\Phi(s, t)$), and consequently t' too, contrary to assumption. Moreover, the values of s' must be the same as those of s on ${}^\bullet e^\bullet$. If $\max\{t'_x \mid x \in {}^\bullet e\} <$

[4]To some readers this may seem like a cheat. To appreciate the idea better, think of proving a property by induction. Sometimes it suffices to carry out the induction with nothing more than the desired property, but sometimes it is necessary to prove more than the desired property in order to carry out all of the inductive steps. Specifying the abstraction invariant requires a similar balance where the relation may need to satisfy more than its basic requirement in order for all of the parts to fit. In this case, it suffices to assert only that the production rule is satisfied.

$\min\{t'_y \mid y \in e^\bullet\}$ then Inequality (1) holds too, so $\mathcal{A}, s' \vDash e$ follows from $\Phi(s, t)$ and the fact that s' has the same values as s on ${}^\bullet e^\bullet$.

To prove that ϕ is an abstraction server for Φ, we must show that the following rule is satisfied:

$$\frac{\Phi(s, t) \quad s \nearrow U = s' \nearrow U \quad \downarrow U \text{ is consistent in } s'}{\Phi(s', t')}$$

where $t' = \phi(U, s', t)$ and where U is a change set. Let e be an event, and suppose that t' is defined on ${}^\bullet e^\bullet$ and

$$\max\{t'_x \mid x \in {}^\bullet e\} < \min\{t'_y \mid y \in e^\bullet\}. \tag{3}$$

We must show that s' is defined on ${}^\bullet e^\bullet$ and $\mathcal{A}, s' \vDash e$. If ${}^\bullet e^\bullet \cap U = \emptyset$, then the values in question are the same as those for s, t, so the desired conclusion follows from the fact that $\Phi(s, t)$ holds. Suppose therefore that ${}^\bullet e^\bullet \cap U \neq \emptyset$. Because U is a change set it cannot contain elements of both ${}^\bullet e$ and $e^\bullet$. Suppose first that $U \cap {}^\bullet e \neq \emptyset$. Then there is a contradiction with Inequality (3) because the values of t' to the right of x must be the same as those of t, but the definition of ϕ says that t'_x is larger than any of these. Second, suppose that $U \cap e^\bullet \neq \emptyset$. Then there is no problem because $\downarrow$ U was assumed to be consistent in s' and $e \in \downarrow U$.

Equation (2) is, of course, different from the dates that would be assigned to modified files by consulting the system clock, so it is not exactly the same as the `make` abstraction server. This underscores the fact that a given abstraction may have many servers that could implement it. It is important for any choice of server to prove that it does indeed satisfy the expected invariant. For example, the server above does not leave one to wonder about the correctness of builds made after a reboot has caused or corrected an error in the time on the system clock. Practical problems with `make` have arisen from failures to respect the precise invariants of the build abstraction. For example, it is assumed above that the dates on the files are set by the abstraction server. If these dates were instead set by another mechanism as the result, for instance, of being copied from a repository, then the invariants would be broken, and the integrity of the build would no longer be ensured.

4.2 A Customized Abstraction

Let us consider the optimization proposed for the p-net in Figure 6. In this example, it seems potentially worthwhile to retain the header `y.tab.h` for comparison to subsequent versions with later dates to avoid recompiling the lexer if no changes have occurred. To describe the abstraction, we refer to the names for p-net elements appearing in Figure 11. Let $\mathcal{A}$ be the *standard* model for this production net as we described it earlier. The abstraction keeps an abstract value for y consisting of a "date" and the

standard value of x that was used to create the standard value of y. If it is found that the standard value of x is unchanged and the date of z is earlier than that of y, then no rebuild is required.

Whenever e is evaluated, the abstraction server needs to reset the date on y and cache the standard value of x on which e was evaluated. In this and some subsequent examples, it will be convenient to write down some partial sections explicitly: if i, j, k are elements of I, and a, b, c lie in S_i, S_j, S_k respectively, then $s = (i, j, k \mapsto a, b, c)$ is the partial section with i, j, k as its domain of existence and with s_i, s_j, s_k respectively equal to a, b, c. Here is the description:

—*Model:* Only variables x, y, z are of interest, so $B^{\mathcal{B}} = \{x, y, z\}$; and only the event e will be tested for the optimization, so $E^{\mathcal{B}} = \{e\}$. We take $V_x^{\mathcal{B}} = V_x^{\mathcal{A}}$ and $V_y^{\mathcal{B}} = V_x^{\mathcal{B}} \times \omega$ and $V_z^{\mathcal{B}} = \omega$. The relation $\mathcal{B}, t \vDash e$ holds if, and only if, t is defined on ${}^\bullet e^\bullet$ and t_y is a pair (u, n) such that $u = t_x$ and $n > t_z$.

—*Abstraction:* The relation $\Phi(s, t)$ holds if, and only if, two conditions hold: (1) if t is defined on x, then so is s and $s_x = t_x$; (2) if t is defined on y, z and $t_y = (u, n)$ where $t_z > n$, then s is defined on y, z and $(x, z \mapsto u, s_z)\ R_e^{\mathcal{A}}\ (y \mapsto s_y)$.

—*Server:* Suppose U is a change set. The value of $t' = \phi(U, s', t)$ is the same as that of t outside of U. If $x \in U$, then $t'_x = s'_x$. If $y \in U$, then $t'_y = (t_x, t_z + 1)$ assuming t is defined on x, z; otherwise t'_y is undefined. If $z \in U$ and $t_y = (u, n)$, then $t'_z = n + 1$, but, if t_y does not exist, then $t'_z = 0$.

We must show that Φ is an abstraction. To prove soundness with respect to production, suppose that $\Phi(s, t)$ and $\mathcal{B}, t \vDash e$. This means that t is defined on ${}^\bullet e^\bullet$, and t_y is a pair (u, n) where $u = t_x$ and $n > t_z$. This means that t satisfies the hypotheses of the second condition for the abstraction, so s is defined on ${}^\bullet e^\bullet$ and $(x, z \mapsto u, s_z)\ R_e^{\mathcal{A}}\ (y \mapsto s_y)$. But t also satisfies the first condition for the abstraction Φ, so $s_x = t_x = u$, which means $\mathcal{A}, s \vDash e$ as desired. Soundness with respect to deletion is straightforward. The proof that ϕ is an abstraction server is omitted; it is a hybrid of the proof above for the date abstraction and the argument below for difference abstractions.

Is this really an "abstraction"? Since old concrete values were kept for comparison with new values, the "abstract" model is not more abstract for such values than the standard one. This terminological foible can be considered in light of the remaining two examples of abstractions considered here.

4.3 Difference Abstraction

The difference abstraction caches the sources that were used to build a target. This information can be used to avoid subsequent rebuilds when sources have not changed. To describe the abstraction precisely, we need some more mathematical notation. Given a product $X \times Y$, let $\mathrm{fst} : X \times Y \to X$ be a projection onto the first coordinate, and let $\mathrm{snd} : X \times Y \to Y$ be a projection onto the second. When working with expressions that may not exist, like s_i where s is a partial section, it is useful to write equations using *Kleene equality*: given expressions P and Q, we write $P \simeq Q$ to mean that (1) P exists if, and only if, Q does, and (2) if P, Q exist then $P = Q$.

—*Model:* Define $B^{\mathcal{B}} = B$ and $E^{\mathcal{B}} = E$. Define

$$V_x^{\mathcal{B}} = \begin{cases} V_x^{\mathcal{A}} & x \text{ a source} \\ V_x^{\mathcal{A}} \times \tilde{\Pi}(V_y^{\mathcal{A}} \mid y \in {}^\bullet e^\bullet) & x \text{ produced by } e \end{cases}$$

$$c(t, x) \simeq \begin{cases} t_x & \text{if } x \text{ is a source} \\ \mathrm{fst}(t_x) & \text{if } x \text{ is a target.} \end{cases}$$

And, for each $\mathcal{B}$-state t and $e \in E$, define

$$(t \mid {}^\bullet e)\ R_e^{\mathcal{B}}\ (t \mid e^\bullet)$$

if, and only if, there is a variable $y \in e^\bullet$ such that

$$(c(t, x) \mid x \in {}^\bullet e^\bullet) = \mathrm{snd}(t_y).$$

—*Abstraction:* For any $\mathcal{A}$-state s and $\mathcal{B}$ state t, the relation $\Phi(s, t)$ holds if, and only if,

(1) for every source x, $t_x \simeq s_x$ and
(2) for every target x, t_x is defined if, and only if, s_x is defined, and, if they are defined, then $t_x = (s_x, (s' \mid {}^\bullet e^\bullet))$ for some $\mathcal{A}$-state s' such that $\mathcal{A}, s \vDash e$.

—*Server:* For any change set $U \subseteq B$ and $\mathcal{A}$-state s' and $\mathcal{B}$-state t define

$$\phi(U, s', t)_x \simeq s'_x$$

if x is a source variable in U, but if x is a target variable in U, define $\phi(U, s', t)_x \simeq (s'_x, (s'_y \mid y \in {}^\bullet e^\bullet))$ where e is the event that produced x. If x is not in U, define $\phi(U, s', t)_x \simeq t_x$.

To show that Φ is sound with respect to production, suppose $\Phi(s, t)$ and $\mathcal{B}, t \vDash e$ for some s, t, e. We must show that $\mathcal{A}, s \vDash e$ also holds. The fact that $\mathcal{B}, t \vDash e$ means that there is a target $y \in e^\bullet$ such that $\mathrm{snd}(t_y)$ is the partial section $u = (c(t, x) \mid x \in {}^\bullet e^\bullet)$. Now, the assumption that $\Phi(s, t)$ holds tells us two things. First, $c(t, x) \simeq s_x$ for each x; this means that u is

$(s \mid {}^\bullet e^\bullet)$. Second, since t_y is defined, it has the form $(s_y, (s' \mid {}^\bullet e^\bullet))$ where $\mathcal{A}$, $s \vDash e$. But, by Lemma 2, these facts imply that $\mathcal{A}$, $s \vDash e$, as desired. That Φ is also sound with respect to deletion is straightforward, noting that the domains of existence of s, t are the same if $\Phi(s, t)$ holds, so the domains of existence of $(s \mid U)$ and $(t \mid U)$ will also be the same for any set of variables U.

To see that ϕ is a server for Φ, let U be a change set of variables; let s be a $\mathcal{A}$-state; and let t be a $\mathcal{B}$-state. Suppose that $\Phi(s, t)$, and suppose that s' is an $\mathcal{A}$-state such that $s \nearrow U = s' \nearrow U$ and $\downarrow U$ is consistent in s'. We must prove that $t' = \phi(U, s', t)$ satisfies $\Phi(s', t')$. First, if x is not in U, then s'_x, t'_x are the same as s_x, t_x; the desired properties hold because $\Phi(s, t)$ does. Suppose $x \in U$. If x is a source, then $t'_x \simeq s'_x$ by definition, and the condition on Φ for sources is therefore satisfied. If, on the other hand, x is produced by e, then $t'_x \simeq (s'_x, (s' \mid {}^\bullet e^\bullet))$. But $e \in \downarrow U$ and $\downarrow U$ is consistent in s'; thus, in particular, $\mathcal{A}$, $s' \vDash e$.

4.4 Fingerprinting Abstraction

The difference abstraction is inefficient in some ways: the abstraction keeps the entirety of the old values used to produce the new ones, and the abstraction condition must check whether this value is equal to new values, possibly many times. To save space and time, it might be worthwhile to save a *compressed* version of the old value and compare this to compressed versions of the new values. We could choose to do the compression in such a way that the compressions of two values are the same if, and only if, the values themselves are the same. That is, we could choose an injective compression map. However, we are not generally interested in *uncompressing* the values in this case, only in keeping enough of a record of the values that an equality test can be carried out efficiently. This leads us naturally to the idea that if the "compression" is *almost* injective, then this will be good enough, because the probability of the "compressions" of two different values being the same is acceptably low. This is the idea behind fingerprinting, as discussed earlier in the context of the SML/NJ Compilation Manager. To fit fingerprinting into the theoretical framework of this article demands that we reconcile the fingerprinting concept of being correct *almost always* with the correctness criteria for abstractions, which stipulates correctness *in all cases*.

Perhaps the simplest way to achieve this reconciliation between correctness and almost-correctness is to focus the uncertainty about correctness in the relation between the *actual* model and an *approximate* model. Let $\mathcal{A}$ be the intended model for a production net $N = (B, E, S, T)$, and suppose α is a build server for $\mathcal{A}$. For each $x \in B$, let us assume we are given a space F_x of "fingerprints" and a fingerprinting function $f_x : V_x^{\mathcal{A}} \to F_x$. We define a new model $\bar{\mathcal{A}}$ as follows. The events and variables of $\bar{\mathcal{A}}$ are the same as those of $\mathcal{A}$, that is, $B^{\bar{\mathcal{A}}} = B^{\mathcal{A}}$ and $E^{\bar{\mathcal{A}}} = E^{\mathcal{A}}$. For each $x \in B$, we define

$V_x^{\bar{\mathcal{A}}} = V_x^{\mathcal{A}} \times F_x$, and for any event e and $\bar{\mathcal{A}}$-state s, we define $(s \mid {}^\bullet e)\ R_e^{\bar{\mathcal{A}}}\ (s \mid e^\bullet)$ if, and only if, there is an $\mathcal{A}$-state s' such that

$$(s' \mid {}^\bullet e)\ R_e^{\mathcal{A}}\ (s' \mid e^\bullet)$$

and, for each $x \in {}^\bullet e^\bullet$, $f_x(s'_x) \simeq \mathrm{snd}(s_x)$. That is, a relation R_e holds in $\bar{\mathcal{A}}$ if, and only if, the values of the prerequisites and targets have fingerprints that could have been obtained from a related set of values in $\mathcal{A}$. *In particular,* if f_x is an injection, then $f_x(s'_x) \simeq \mathrm{snd}(s_x) \simeq f_x(\mathrm{fst}(s_x))$, so $s'_x = \mathrm{fst}(s_x)$. If f_x is an injection for each $x \in B$, then $\mathcal{A}$ and $\bar{\mathcal{A}}$ are isomorphic. A server for $\bar{\mathcal{A}}$ is also needed. For any e, s, x, define $\bar{\alpha}(e, s)_x = (\alpha(e, s)_x, f_x(\alpha(e, s)_x))$.

This formulation shows that the fidelity of $\bar{\mathcal{A}}$ to $\mathcal{A}$ is directly related to properties of the fingerprinting function. For instance, if most changes are likely to be small, then it is important that similar files very rarely or never have the same fingerprint. Certain classes of changes, like copying between files (for instance, to move a procedure declaration into a shared library) should also cause fingerprints to change.

We are now prepared to describe fingerprinting as an abstraction of the approximate model $\bar{\mathcal{A}}$.

—*Model:* Define $B^{\mathcal{B}} = B$ and $E^{\mathcal{B}} = E$. Define

$$V_x^{\mathcal{B}} = \begin{cases} F_x & \text{if } x \text{ a source} \\ F_x \times \tilde{\Pi}(F_y \mid y \in {}^\bullet e^\bullet) & \text{if } x \text{ is a target.} \end{cases}$$

$$c(t, x) \simeq \begin{cases} t_x & \text{if } x \text{ is a source} \\ \mathrm{fst}(t_x) & \text{if } x \text{ is a target.} \end{cases}$$

And, for each $\mathcal{B}$-state t and $e \in E$, define

$$(t \mid {}^\bullet e)\ R_e^{\mathcal{B}}\ (t \mid e^\bullet)$$

if, and only if, there is some $y \in e^\bullet$ such that

$$(c(t, x) \mid x \in {}^\bullet e^\bullet) = \mathrm{snd}(t_y).$$

—*Abstraction:* For any $\bar{\mathcal{A}}$-state s and $\mathcal{B}$-state t, the relation $\Phi(s, t)$ holds if, and only if,

(1) for every source x, $t_x \simeq f_x(s_x)$, and
(2) for every target x, t_x is defined if, and only if, s_x is defined, and, if they are defined, then $t_x = (\mathrm{snd}(s_x), (f_y(s'_y) \mid y \in {}^\bullet e^\bullet))$ for some $\mathcal{A}$-state s' such that $\mathcal{A}, s' \vDash e$.

—*Server:* For any change set $U \subseteq B$ and $\bar{\mathcal{A}}$-state s' and $\mathcal{B}$-state t define

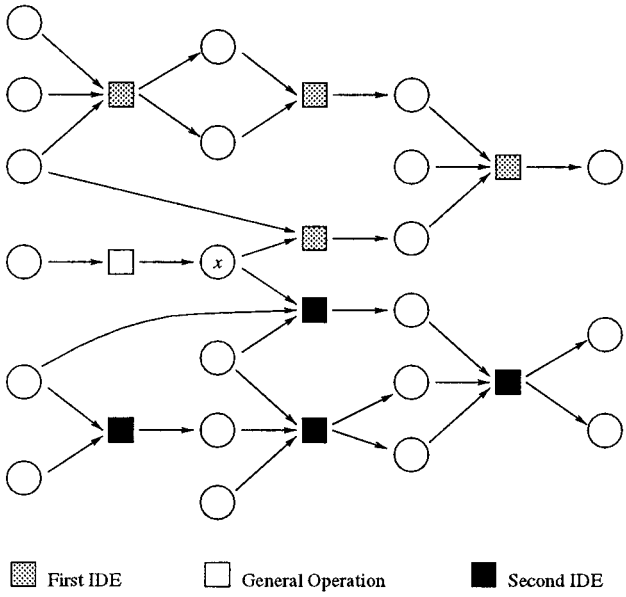

Fig. 13. Integrating models.

$$\phi(U, s', t)_x \simeq \mathrm{snd}(s'_x)$$

if x is a source in U. If x is a target variable in U define

$$\phi(U, s', t)_x \simeq (\mathrm{snd}(s'_x), (\mathrm{snd}(s'_y) \mid y \in {}^\bullet e^\bullet)).$$

If x is not in U, then $\phi(U, s', t)_x \simeq t_x$.

The proofs that Φ is an abstraction and ϕ is a server for it are very similar to the ones given for the difference abstraction above (modulo the tedium of some projections and fingerprintings).

5. COMBINING AND COMPARING MODELS AND ABSTRACTIONS

Suppose we are given a pair of IDEs and a project that needs to manipulate items produced by them. Each of the IDEs controls the dependencies, abstractions, and build operations for a portion of the collection of items used by the project. Figure 13 illustrates the general idea. The IDEs supply the servers that are used in the overall build, which is controlled by computation over the underlying p-net according to the rules in Figure 12. Semantics-dependent abstractions may be supplied by the IDEs, and semantics-independent abstractions such as the `make` date abstraction may be available in the general build environment. How can we combine these models and abstractions to yield an overall build taking advantage of all of the abstractions together? When models and abstractions "overlap" by applying to common variables or events, what consistency requirements

would we expect to hold? Can we find a way to compare two abstractions to determine if one of them is more "complete" than the other (because it "more often" recognizes the possibility of omitting a build step)?

In this section we look at each of these questions and provide a rigorous formulation. The main idea is to generalize the semantics of a build to allow for multiple abstractions. We then show how to merge models and extend abstractions to the merged models. This will allow a collection of models and corresponding abstractions to be merged into a single model with multiple abstractions and then built using the generalized multiple-abstraction semantics. The multiple-abstractions generalization can also be used to define a way to compare two abstractions on the same model. This comparison technique is used to show that difference abstraction is more complete than date abstraction.

5.1 Multiple Abstractions

Suppose we have a model $\mathcal{A}$ and abstractions $\Phi : \mathcal{A} \to \mathcal{B}$ and $\Phi' : \mathcal{A} \to \mathcal{B}'$. What do we want to do to combine the two abstractions? Remember the purpose of an abstraction: it is to help recognize when a build does not need to take place because the desired relationships already hold. This means that a pair of abstractions should be used disjunctively. There are two possibilities about how to proceed, each having its minor technical drawbacks. Perhaps the simplest approach conceptually is to define some form of "disjunction" operation $\vee$ for the abstractions Φ, Φ' and a product $\times$ for the models $\mathcal{B}$, $\mathcal{B}'$, and then define a new combined abstraction $\Phi \vee \Phi' : \mathcal{A} \to \mathcal{B} \times \mathcal{B}'$. The model $\mathcal{B} \times \mathcal{B}'$ is basically an indexed family of pairs with components drawn from each of the two models. Details are omitted because we instead pursue an alternate formulation which, although more complicated to define, is easier to work with in examples.

The idea is to combine abstractions not by creating a combined abstraction, but rather by allowing multiple abstractions to be part of the operational semantics of the build. This entails generalizing the rules in Figure 12 to allow a list of abstraction states. Build states have the form $M, s, t_1, \ldots, t_n$ where M is a marking; s is a state of the standard model $\mathcal{A}$; and the states $t_1, \ldots, t_n$ are from abstract models $\mathcal{B}_1, \ldots, \mathcal{B}_n$. The omission rule seeks any instance in which at least one of these abstractions indicates that the build is unnecessary. Even if the build is unnecessary, the abstractions that did not apply are updated. Figure 14 gives the details. The soundness result for the rules in Figure 14 is the following generalization of Theorem 3:

THEOREM 4. *Let $\Phi_i : \mathcal{A} \to \mathcal{B}_i$ for $i = 1, \ldots, n$ be abstractions between models of a p-net $N = (B, E, S, T)$. Suppose s is a state of $\mathcal{A}$, and t_i are states of $\mathcal{B}_i$ for $i = 1, \ldots, n$ such that $\Phi_i(s, t_i)$. Let M be a consistent marking of $\mathcal{A}$, s. If $M, s, t_1, \ldots, t_n \to^* M', s', t'_1, \ldots, t'_n$ with respect to a build server α for $\mathcal{A}$ and abstraction servers ϕ_i for Φ_i, then*

$$\text{Initiation} \quad \frac{M/e \qquad \mathcal{B}_i, t_i \not\models e \text{ for all } i}{M, s, t_1, \ldots, t_n \xrightarrow{\hat{e}} M \cup \{e\}, s, t_1, \ldots, t_n}$$

$$\text{Termination} \quad \frac{e/M \qquad s' = \alpha(e, s)}{M, s, t_1, \ldots, t_n \xrightarrow{\check{e}} M \cup e^\bullet, s', \phi_1(e^\bullet, s', t_1), \ldots, \phi_n(e^\bullet, s', t_n)}$$

$$\text{Omission} \quad \frac{M/e \qquad \mathcal{B}_i, t_i \models e \text{ for some } i}{M, s, t_1, \ldots, t_n \xrightarrow{\epsilon} M \cup \{e\} \cup e^\bullet, s, t'_1, \ldots, t'_n}$$

where $t'_j = t_j$ if $\mathcal{B}_j, t_j \models e$ and $t'_j = \phi_j(e^\bullet, s', t_j)$ otherwise.

Fig. 14. Computation relative to servers α and $\phi_1, \ldots, \phi_n$.

(1) *M' is a consistent marking of $\mathcal{A}$, s' and*

(2) *$\Phi_i(s', t'_i)$ for each i.*

The proof is basically the same as that of Theorem 3.

5.2 Merging Models and Extending Abstractions

While we have now dealt with how to combine multiple abstractions over a common model, the implication of Figure 13 is that we may not be starting from a common model. Given a collection of models and abstractions over a given production net, how should we combine them to take advantage of the operational semantics in Figure 14? The key observation is that a pair of models must *agree* on the parts of the p-net where they overlap. If they do not agree, then it will make no sense to merge them; if they do agree then merging them consists basically of taking their union. Here is the formal definition:

Definition. Let $\mathcal{A}_1 = (B^1, E^1, V^1, R^1)$ and $\mathcal{A}_2 = (B^2, E^2, V^2, R^2)$ be models of a net $N = (B, E, S, T)$. They are said to be *mergeable* if, for each $x \in B^1 \cap B^2$, the sets V^1_x and V^2_x are the same, and, for each $e \in E^1 \cap E^2$, the relations R^1_e and R^2_e are also the same.

If they are mergeable, their *merge* $\mathcal{A}_1 \cup \mathcal{A}_2$ is a model $\mathcal{A} = (B, E, V, R)$ over N where $B = B_1 \cup B_2$ and $E = E_1 \cup E_2$. For each $x \in B$, define

$$V_x = \begin{cases} V^1_x & \text{if } x \in B_1 \\ V^2_x & \text{if } x \in B_2. \end{cases}$$

For each $e \in E$ define $\mathcal{B}$, $s \models e$ if, and only if, $e \in E_1$ and B_1, $(s \mid B^1) \models e$ or $e \in E_2$ and $\mathcal{B}_2$, $(s \mid B^2) \models e$.

For instance, the IDE models in Figure 13 are disjoint on events; if they also agree on the sets they assign to variables (some of which may be shared as prerequisites or targets of events from both models) then they are mergeable.

Suppose finally now that we are given two mergeable models and abstractions for each model. Specifically, suppose $\mathcal{A}_1 = (B_1, E_1, V_1, R_1)$ and $\mathcal{A}_2 = (B_2, E_2, V_2, R_2)$ are mergeable models of a net N and $\Phi : \mathcal{A}_1 \rightarrow \mathcal{B}$ is an abstraction over $\mathcal{A}_1$. The *extension* of Φ to $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ is the relation Ψ between $\mathcal{A}$-states and $\mathcal{B}$-states defined by taking $\Psi(s, t)$ to hold if, and only if, $\Phi((s \mid B_1), t)$ does.

LEMMA 5. *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are mergeable models of a net N and $\Phi : \mathcal{B} \rightarrow \mathcal{A}_1$ is an abstraction over $\mathcal{A}_1$, then the extension $\Psi : \mathcal{A}_1 \cup \mathcal{A}_2 \rightarrow \mathcal{B}$ of Φ to the merge of $\mathcal{A}_1$ and $\mathcal{A}_2$ is also an abstraction.*

PROOF. First, it is clear that the necessary inclusions hold between the sets of variables and events of $\mathcal{B}$ and those of $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$. If $\Psi(s, t)$ and $\mathcal{B}$, $t \vDash e$, then $\Phi((s \mid B_1), t)$ by definition of the extension, so A, $(s \mid B_1) \vDash e$. Since ${}^\bullet e^\bullet \subseteq B_1$ by the definition of an abstraction, it follows from Lemma 2 that A, $s \vDash e$. Thus the production rule is satisfied. Since the restriction of s is also a restriction of $(s \mid B_1)$, the deletion rule is also satisfied. □

It is also possible to show that an abstraction server for Φ is also an abstraction server for any of its extensions.

As a way of summarizing, let us now go through the steps required to set up a possible operational semantics of the p-net in Figure 13. Suppose $\Phi_1 : \mathcal{A}_1 \rightarrow \mathcal{B}_1$ is the model and an abstraction for the events and variables of the first IDE, and $\Phi_2 : \mathcal{A}_2 \rightarrow \mathcal{B}_2$ is a similar model and abstraction for events and variables of the second IDE. Let us assume that $\mathcal{A}_3$ is a model of variables and events that are not part of the two IDEs. Now, to have a consistent configuration, it must be the case that $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ all agree on whatever variables they have in common. For instance, the variable x in the p-net there is shared by all three models, so we must have $V_x^{\mathcal{A}_1} = V_x^{\mathcal{A}_2} = V_x^{\mathcal{A}_3}$. Assuming that this is indeed the case, the models can be merged to form a common model $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3$. The abstractions Φ_1 and Φ_2 can now be extended to abstractions $\Psi_i : \mathcal{A} \rightarrow \mathcal{B}_i$, for $i = 1,2$ respectively, for this larger model, which covers all of the variables and events of the p-net in Figure 13. A generic form of abstraction, like date abstraction, can now be applied to the p-net as a whole, or to any part of it (such as the parts where the models overlap). Let $\Psi_3 : \mathcal{A} \rightarrow \mathcal{B}_3$ be such an abstraction. The operational semantics of the resulting p-net is obtained by applying the rules of Figure 14 to tuples of the form M, s, t_1, t_2, t_3 where M is a marking of the underlying p-net; s_1 is a state of the standard model $\mathcal{A}$; and t_i is a state of $\mathcal{B}_\mathcal{J}$, the target of the abstraction Ψ_i for each $i = 1,2,3$.

5.3 Comparing Abstractions

Suppose we are using more than one abstraction as part of a build. Is it possible that one of the abstractions is superfluous because any omission

that it would find would be found by one of the others anyway? A more general question is can we classify abstractions according to how "strong" they are, so that weaker abstractions do not add power to stronger ones? Let us now formulate a concept like this and use it to show that there is a sense in which date abstraction is *weaker* than a general class of abstractions. Here is the main definition:

Definition. Suppose $\Phi, \phi : \mathscr{A} \to \mathscr{B}$ and $\Psi, \psi : \mathscr{A} \to \mathscr{C}$ are abstractions. Define a relation $\preceq$ so that $\Phi, \phi \preceq \Psi, \psi$ holds if, and only if, the following invariant is satisfied: if the following four conditions hold,

(1) $\Phi(s, t)$ and $\Psi(s, u)$,

(2) for every event e, $\mathscr{B}, t \vDash e$ implies $\mathscr{C}, u \vDash e$,

(3) M is a consistent marking of $\mathscr{A}, s$, and

(4) $M, s, t, u \to^* M', s', t', u'$,

then, for every event e, $\mathscr{B}, t' \vDash e$ implies $\mathscr{C}, u' \vDash e$.

If $\Phi, \phi \preceq \Psi, \psi$ we say that Φ, ϕ is *weaker* than Ψ, ψ. Intuitively, an abstraction is weaker than another one if there is "no difference" between running both together and running only the stronger one.

Before looking at the weakness relation on examples, let us formulate another concept about the strength of an abstraction:

Definition. An abstraction $\Phi, \phi : \mathscr{A} \to \mathscr{B}$ is said to be *progressive* if

$$\mathscr{B}, \phi(e^\bullet, s', t) \vDash e$$

whenever $\mathscr{A}, s' \vDash e$ and s' is defined on at least one variable in $e^\bullet$.

Intuitively, this says that the build server "captures" all of the instances in which the rebuild does not need to occur assuming at least one target was created. One might think at first that any server should do at least this much, but, in fact, the *date* abstraction is not progressive. To see this, consider the p-net shown in Figure 15 and suppose that a correct build on e results in the creation of a value for y but not for z. Since the date abstraction requires all targets to be defined, the abstraction server will dutifully update the abstract value of y, on which the standard state is defined, but not of z. Intuitively, a rebuild must assume that z has been deleted, when, in fact, it was never created. It is possible to augment the definition of date abstraction to provide a progressive abstraction by causing each output abstract value to retain information about existence of the others (the author is not aware of any SCM system that does this). On the other hand, the difference abstraction does keep the necessary information:

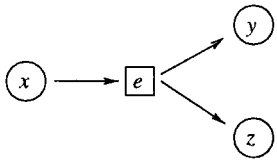

Fig. 15. P-net example.

PROPOSITION 6. *Difference abstraction is progressive. That is, for any model* $\mathcal{A}$*, if* $\Phi, \phi : \mathcal{A} \to \mathcal{B}$ *is the difference abstraction on* $\mathcal{A}$ *with its usual server, then* Φ, ϕ *is progressive.*

PROOF. Suppose $\mathcal{A}, s' \vDash e$ and s' is defined on $x \in e^\bullet$: is it the case that $\mathcal{B}, \phi(e^\bullet, s', t) \vDash e$? By definition, $\phi(e^\bullet, s', t)_x = (s'_x, (s'_y \mid y \in {}^\bullet e^\bullet))$. The definition of the difference abstraction says that the relation in $\mathcal{B}$ holds if the second coordinate of this value is a "snapshot" of s' on ${}^\bullet e^\bullet$, but this is exactly how the abstraction server value was defined. □

The relation $\leq$ is a preordering but not a partial ordering. That is, the relation is transitive and reflexive, but the fact that $\Phi, \phi \leq \Psi, \psi$ and $\Psi, \psi \leq \Phi, \phi$ both hold does not mean that $\Phi = \Psi$ or $\phi = \psi$.

We can now state our general result about date abstraction.

THEOREM 7. *Date abstraction is weaker than any progressive abstraction. That is, if* $\mathcal{A}$ *is a model with date abstraction* $\Phi, \phi : \mathcal{A} \to \mathcal{B}$ *and progressive abstraction* $\Psi, \psi : \to \mathcal{C}$*, then* $\Phi, \phi \leq \Psi, \psi$.

PROOF. Assume that the four conditions in the definition of $\leq$ hold for Φ and Ψ. The proof is by induction on the length of the relation $M, s, t, u \to^* M', s', t', u'$. We must show that, for any single step, $\hat{e}, \check{e}, \epsilon$, the desired invariant holds, i.e., $\mathcal{B}, t' \vDash e$ implies $\mathcal{C}, u' \vDash e$. This is obvious for transitions $\hat{e}$ and ϵ because neither of them changes the abstract state (t, u), so we need only concern ourselves with the instance:

$$\frac{e/M \qquad s' = \alpha(e, s)}{M, s, t, u \xrightarrow{\check{e}} M \cup e^\bullet, s', \phi(e^\bullet, s', t'), \psi(e^\bullet, s', u')}$$

We need only concern ourselves with events f where $\mathcal{B}, t' \vDash f$ and $\mathcal{C}, u' \nvDash f$ or $\mathcal{B}, t' \nvDash f$ and $\mathcal{C}, u' \vDash f$. All others satisfy the desired invariant by assumption. Thus we need to consider e together with any f such that ${}^\bullet f \cap e^\bullet \neq \emptyset$. Now, we have $\mathcal{C}, u' \vDash e$ because Ψ, ψ is progressive, so there is no problem with e. Consider f such that ${}^\bullet f \cap e^\bullet \neq \emptyset$. Since $f \in \uparrow \{x\}$ for some $x \in e^\bullet$, Eq. (2) implies that Inequality (1) *fails* for the prerequisites and targets of f. That is, $\mathcal{B}, t' \nvDash f$. Hence, it is vacuously true that the desired invariant is satisfied. □

Corollary 8. Date abstraction is weaker than difference abstraction. That is, if $\mathcal{A}$ is a model with date abstraction Φ, $\phi : \mathcal{A} \to \mathcal{B}$ and difference abstraction Ψ, $\psi : \to \mathcal{C}$, then $\Phi \leq \Psi$.

Does this mean difference abstraction is "better" than data abstraction? Although being stronger is advantageous in avoiding build steps, the overhead of maintaining the abstraction is also significant. For date abstraction, the overhead of creating and checking dates is comparatively small, so despite its weakness it is an effective abstraction.

6. RELATED WORK AND CONCLUSIONS

The accomplishments of this article are the introduction of production nets and their models, the formulation of abstractions and their associated correctness conditions, the application of these concepts in a collection of noteworthy cases, and the formulation of the concepts of composition and strength of abstractions. It may be possible to use this formulation as a foundation for building SCM tools that provide an extensible abstraction mechanism capable of automating both a range of useful semantics-independent optimizations like difference abstraction and a collection of semantics-dependent optimizations drawn from the domain of the system being built.

This article has analyzed only the correctness properties of build abstractions, but the model presented here could also be used as a basis from which to analyze the potential efficiency gain of a build optimization as a balance between the overhead of maintaining the abstraction and the savings from finding that a rebuild is unnecessary. An experimental analysis of this kind for a semantics-dependent abstraction appeared in Tichy [1986], but the existence of a general model of evaluation opens the possibility of a theoretical analysis as well. In practice or in theory, performance gains at least as good as those of Tichy [1986] seem likely for applications such as the HOL abstraction sketched in Section 2. The main problem is to provide a convenient mechanism for programming such abstractions and adding them modularly to an SCM tool.

While the abstraction concept introduced here covers a wide range of interesting cases it is an open question how to extend it to more general dependency structures. P-nets can probably be extended to deal with conditional builds. On the other hand, cycles, which are explicitly ruled out by the restriction of p-nets to be acyclic graphs, will be more difficult to accomodate. Some multipass systems, such as the typesetting program LaTeX, allow apparent cycles in builds and use optimizations to recognize when a pass can be omitted. It is unobvious how to treat this with p-nets simply by "unwinding" the cycles. The Vesta system [Levin and McJones 1993] has a theoretical basis [Abadi et al. 1996] for dealing with cycles for a general caching optimization, and this may provide guidance.

There is no end to the task of finding new build optimizations. At heart it is a general question of incremental computation: given a function f, is

there a good way to tell if $f(x)$ is the same as $f(x')$ if you know something about how x and x' differ? Whether an answer to this question is interesting depends on the cost of this determination, so pragmatics will be the key concern. General work on incremental computation such as that of Liu and Teitelbaum [1995] and Liu [1996] is, therefore, potentially useful for configuration maintenance if common application cases can be found. An SCM system that provides extensible abstractions would be a good way to realize these opportunities.

As mentioned before, Borison's [1987] work on software manufacture includes a definition of a graph structure almost the same as p-nets. Luqi [1990] also uses these structures, but for modeling software project evolution rather than builds. Borison's objectives are similar to those of this article. She introduces the concept of a *difference predicate* to model cases in which a change does not require a rebuild. For instance, a difference predicate on single items is a function from pairs of "components" to booleans indicating when one component can be substituted for another without the need for a rebuild. Perhaps the easiest way to understand the relationship between abstractions and difference predicates is to note that abstractions define difference predicates by describing an operational semantics for them. A topic considered in Borison [1987] that was not considered here is the definition of subgraphs representing separately manufactured subsystems. This topic would be of interest for p-nets and abstractions, since it would provide a refinement theory for configurations.

One thing that was not covered in the section on applications of abstractions was the formulation of correctness criteria for various "smart compilation" strategies. The correctness of such abstractions is intimately dependent upon the semantics of the programming language involved. Tichy [1986] states the kind of result one would like to prove, but is unable to provide an actual mathematical proof without a precise mathematical formulation of the semantics of the programming language whose builds are being optimized. Work like that of Cardelli [1997] on the semantics of linking could perhaps be applied to formulating and proving correctness for "smart" compilation.

In another line of work on a rigorous treatment of SCM structures, Zeller [1995] has shown how feature structures, an algebraic construct from computational linguistics, can be used to model a variety of basic operations. The focus of this work is primarily on version management rather than build optimization.

At the current time no SCM tool the author is aware of employs a method for creating and merging abstractions as general as the mathematics this article describes. Some tools do provide hooks for domain-specific extensions for automated dependency analysis. Further support of extensible build optimizations could be a worthwhile step toward merging the flexibility advantages of an open SCM tool with the domain sensitivity of an IDE.

APPENDIX

Some equations are helpful in working with the p-net orderings. Left-closed subsets of $\sqsubseteq_N$ are our principal concern. Let e be an event, then

$$\downarrow\{e\} = \{e\} \cup \downarrow {}^\bullet e. \tag{4}$$

To see this, suppose first that $x \in \downarrow\{e\}$, that is, $x \sqsubseteq_N e$. This means $x = e$ or that there is some y such that $x \sqsubseteq_N y$ and $y\ S\ e$. In the former case x is in $\{e\}$, and in the latter case it is in $\downarrow {}^\bullet e$ so $\subseteq$ holds. Suppose, on the other hand, that $x \in \{e\} \cup \downarrow {}^\bullet e$. If $x = e$ then $x \sqsubseteq_N e$. If $x \in \downarrow {}^\bullet e$, then there is some $y \in {}^\bullet e$ such that $x \sqsubseteq_N y$, so $x \sqsubseteq_N e$ too; and therefore $\supseteq$ holds, proving the equation. Another useful equation describes the left-closure of a collection of targets:

$$\downarrow e^\bullet = e^\bullet \cup \downarrow\{e\} \tag{5}$$

This equation depends on the unique-producer condition; indeed it is a key reason for imposing that condition on production nets. To prove it, suppose $x \in \downarrow e^\bullet$. Then $x \sqsubseteq_N y$ for some $y \in e^\bullet$, so $x = y$; or there is an e' such that $x \sqsubseteq_N e'$ and $e'\ T\ y$. If $x = y$, then $x \in e^\bullet \subseteq e^\bullet \cup \downarrow\{e\}$. If the other property holds, then $e' = e$ by unique-producer condition for y, so $x \sqsubseteq_N e$, which means $x \in \downarrow\{e\} \subseteq e^\bullet \cup \downarrow\{e\}$. Thus we can conclude that $\subseteq$ holds. That $\supseteq$ holds follows immediately from the fact that $e^\bullet$ is nonempty and e is in its left-closure. Combining Eq. (4) with an expansion of $\downarrow\{e\}$ using Eq. (5) yields a third equation:

$$\downarrow e^\bullet = e^\bullet \cup \{e\} \cup \downarrow {}^\bullet e \tag{6}$$

which will be useful in proving Lemma 1.

PROOF OF LEMMA 1. The proof of the first two implications is very similar to that of the third, so they are omitted. To prove the third implication, we must show that $M' = M \cup \{e\} \cup e^\bullet$ is both left-closed and target-closed. For the former, we can calculate as follows:

$$\begin{aligned}
\downarrow(M \cup \{e\} \cup e^\bullet) &= \downarrow M \cup \downarrow\{e\} \cup \downarrow e^\bullet \\
&= M \cup \downarrow\{e\} \cup \downarrow e^\bullet && (M \text{ left-closed}) \\
&= M \cup (\{e\} \cup \downarrow {}^\bullet e) \cup (e^\bullet \cup \{e\} \cup \downarrow {}^\bullet e) && (\text{Eqs. (4) and (6)}) \\
&= M \cup \{e\} \cup e^\bullet \cup \downarrow {}^\bullet e \\
&= M \cup \{e\} \cup e^\bullet && (\downarrow {}^\bullet e \sqsubseteq M)
\end{aligned}$$

To prove that M' is target-closed, let f be an event, and suppose $f^\bullet \cap (M \cup \{e\} \cup e^\bullet) \neq \emptyset$. There are two possibilities for the intersection of $f^\bullet$

and $e^\bullet$. If $f^\bullet \cap e^\bullet = \emptyset$, then $f^\bullet \cap M \neq \emptyset$, so $f^\bullet \subseteq M$ because M was assumed to be target-closed. On the other hand, if $f^\bullet \cap e^\bullet \neq \emptyset$, then $f = e$ by the unique-producer condition, so $f^\bullet = e^\bullet \subseteq M'$ as desired. Since it is obvious that $e \searrow M'$, the proof is complete. □

PROOF OF THEOREM 3. The proof is by induction on the length of the evaluation from M, s, t to M', s', t'. If this length is zero then there is nothing to prove. Assume that the length is n and that the result is known for shorter lengths. There are three cases for the the last step, depending on the label to the transition or, equivalently, the instance of the rule applied.

Suppose the last step is

$$M'', s', t' \xrightarrow{\hat{e}} M'' \cup \{e\}, s', t'$$

where $M' = M'' \cup \{e\}$. Then we know that this step is an instance of the initiation rule. Here is what we know based on the hypotheses of that rule (RH) and the inductive hypothesis (IH) on the relation:

$\Phi(s't')$	IH
M''/e	RH
M'' is consistent	IH

That $\Phi(s', t')$ is immediate in this case. Implication (1) of Lemma 1 and the second of these facts imply that M' is a marking. M' must be consistent, since M'' is consistent and since the variables in M' are the same as those in M''.

Suppose that the last step is

$$M'', s'', t'' \xrightarrow{\check{e}} M'' \cup e^\bullet, s', \phi(e^\bullet, s', t'')$$

where $s' = \alpha(s'')$ and $M' = M'' \cup e^\bullet$ and $t' = \phi(e^\bullet, s', t'')$. The last step must come from an instance of the termination rule. Here is what we know:

e/M''	RH
M'' is consistent	IH
$\mathcal{A}, s' \vDash e$	α is a build server
$s' \nearrow e^\bullet = s'' \nearrow e^\bullet$	α is a build server
$\Phi(s'', t'')$	IH

The first fact, together with implication (2) of Lemma 1, implies that M' is a marking. To see that it is consistent, note that if a variable is in M', then it is either in M'' or in $e^\bullet$. Producers of events in M'' are consistent in s' because they are consistent in s'', and by the fourth fact above, these two

states do not differ on the prerequisites and targets of the events in M''. By the unique-producer condition, the producer of an element of $e^\bullet$ must be e, and the consistency of e in s' is the third fact above. We must show also that $\Phi(s', t')$. To this end, consider the set $\downarrow e^\bullet$. It is consistent because it is a subset of M''. The fourth and fifth facts above, together with this observation, lead us to the conclusion of the rule for the build server ϕ, that is, $\Phi(s', \phi(e^\bullet, s', t))$.

Suppose, finally, that the last step is

$$M'', s', t' \xrightarrow{e} M'' \cup \{e\} \cup e^\bullet, s', t'$$

where $M' = M'' \cup \{e\} \cup e^\bullet$. This comes from an instance of the omission rule, and we have the following facts:

e/M''	RH
$\mathcal{B}, t' \vDash e$	RH
$\Phi(s', t')$	IH
M'' is consistent	IH

We are immediately given the conclusion that $\Phi(s', t')$. Implication (1) of Lemma 1 implies that M' is a marking. To see that M' is consistent, suppose x is a variable in M'. If $x \in M''$, then the event that produced it must be consistent in s', since M'' is. If, on the other hand, $x \in e^\bullet$, then the unique-producer condition tells us that e produced x. Now, $\mathcal{B}, t' \vDash e$, so the fact that t' is an abstraction of s' tells us that $\mathcal{A}, s' \vDash e$. So e is consistent, and hence M' is consistent. □

ACKNOWLEDGMENTS

A number of people influenced this work. I especially thank Mahesh V. for his ideas on multiple abstractions and Corollary 5.3. The article also benefited from anonymous referee reports and discussions with the following people: Sandip Biswas, Luca Cardelli, Elsa Gunter, Allan Heydon, Tony Hoare, Michael Jackson, Trevor Jim, Cliff Jones, Dave MacQueen, V. Mahesh, Benli Pierce, Andy Pitts, John Reppy, Sam Weber, and Glynn Winskel.

REFERENCES

ABADI, M., LAMPSON, B., AND LÉVY, J.-J. 1996. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional programming* (ICFP '96, Philadephia, PA, May 24–26), R. Harper and R. L. Wexelblat, Eds. ACM Press, New York, NY, 83–91.

ANSI/IEEE. 1987. ANSI/IEEE standard 1042-1987. Guide to software configuration management. ANSI, New York, NY.

BLUME, M. 1998. CM: A compilation manager.

BORISON, E. 1987. A model of software manufacture. In *Proceedings of the IFIP International Workshop on Advanced Programming Environments* (Trondheim, Norway, June 16–18), R. Conradi, T. M. Didriksen, and D. H. Wanvik, Eds. Springer-Verlag, Berlin, Germany, 197–220.

BRODER, A. 1993. Some applications of Rabin's fingerprinting method. In *Sequences II: Methods in Communication, Security, and Computer Science*, R. M. Capocelli, A. De Santis, and U. Vaccarro, Eds. Springer-Verlag, Vienna, Austria.

BURN, G. L., HANKIN, C., AND ABRAMSKY, S. 1986. Strictness anlaysis for higher-order functions. *Sci. Comput. Program. 7*, 3, 249–278.

CARDELLI, L. 1997. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '97, Paris, France, Jan. 15–17, 1997), P. Lee, Ed. ACM Press, New York, NY, 266–277.

COTTAM, I. D. 1984. The rigorous development of a system version control program. *IEEE Trans. Softw. Eng. SE-10*, 2 (Mar.), 143–154.

DURASOFT. 1993. The Revision Control Engine. (Software). DuraSoft GmbH, Karlsruhe, Germany. http: //wwwipd.ira.uka.de/~RCE.

FELDMAN, S. I. 1978. Make: A program for maintaining computer programs. Computer Science Tech. Rep. 57. AT&T Bell Laboratories, Inc., Murray Hill, NJ. Also available in Softw. Pract. Exp. 9 (1979), pp. 255–265.

FOWLER, G. 1990. A case for make. *Softw. Pract. Exper. 20*, S1 (June 1990), 35–46.

GUNTER, E. L. 1988. Building HOL90 easily everywhere. In *Proceedings of the 11th International Conference on Theorem Provers for Higher-Order Logics* (Canberra, Australia, Sept.),

HANNA, C. B. AND LEVIN, R. 1993. The Vesta language for configuration management. Tech. Rep. 107. Digital Systems Research Center.

IEEE. 1990. IEEE Standard 828-1990: Standard for software configuration management plans. IEEE, New York, NY.

LEVIN, R. AND MCJONES, P. R. 1993. The Vesta approach to precise configuration of large software systems. Tech. Rep. 105. Digital Systems Research Center.

LIU, Y. A. 1996. Incremental computation: A semantics-based systematic transformational approach. Ph.D. Dissertation. Department of Computer Science, Cornell University, Ithaca, NY. Also appeared as 1995 Tech. Rep. TR 95-1551.

LIU, Y. A. AND TEITELBAUM, T. 1995. Systematic derivation of incremental programs. *Sci. Comput. Program. 24*, 1 (Feb. 1995), 1–39.

LUQI. 1990. A graph model for software evolution. *IEEE Trans. Softw. Eng. 16*, 8 (Aug. 1990), 916–927.

ORAM, A. AND TALBOTT, S. 1991. *Managing Projects with Make*. 2nd ed. O'Reilly and Associates.

PERRY, D. E. 1987. Version control in the Inscape environment. In *Proceedings of the 9th International Conference on Software Engineering* (Monterey, CA, Mar. 30–Apr. 2), W. E. Riddle, Ed. IEEE Computer Society Press, Los Alamitos, CA, 142–149.

ROCHKIND, M. J. 1975. The source code control system. *IEEE Trans. Softw. Eng. SE-1*, 4 (Dec.), 364–370.

SCHWANKE, R. W. AND KAISER, G. E. 1988. Smarter recompilation. *ACM Trans. Program. Lang. Syst. 10*, 4 (Oct. 1988), 627–632.

SHAO, Z. AND APPEL, A. W. 1993. Smartest recompilation. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '93, Charleston, SC, Jan. 10–13), S. L. Graham, Ed. ACM Press, New York, NY, 439–450.

TICHY, W. F. 1985. RCS: A system for version control. *Softw. Pract. Exper. 15*, 7 (July 1985), 637–654.

TICHY, W. F. 1986. Smart recompilation. *ACM Trans. Program. Lang. Syst. 8*, 3 (July 1986), 273–291.

ZELLER, A. 1995. A unified version model for configuration management. *SIGSOFT Softw. Eng. Notes 20*, 4 (Oct. 1995), 151–160.

Received: June 1997; revised: October 1997 and November 1998; accepted: June 1999