Abstracting Dependencies between Software Configuration Items

Carl A. Gunter
University of Pennsylvania
http://www.cis.upenn.edu/~gunter

Abstract

This paper studies an abstract model of dependencies between software configuration items based on a theory of concurrent computation over a class of Petri nets. The primary goal is to illustrate the descriptive power of the model and lay theoretical groundwork for using it to design software configuration maintenance tools or model software configurations. As a start in this direction, the paper analyzes and addresses certain limitations in make description files using a form of *abstract interpretation.*

1 Introduction

A variety of formalisms have been created to aid phases of the software engineering life cycle. For instance, logical languages such as Z can be used to describe functional specifications, while structures like flow charts and Petri nets are useful in detailed design. Automated formal verification has been shown feasible in certain cases where system behavior can be described using a finite state machine. However, less attention has been directed at the application of abstract models to the *maintenance* aspects of software. The aim of this work is to study one aspect of software maintenance from this perspective. The approach advocated here is based on ideas and structures from the formal semantics of *concurrency* but adapts them to the particular goals arising in software configuration maintenance.

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 or the 'environment' in some general sense, 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 'product' 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 [4]. 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. An example of a make description file for a small configuration of C-programming files appears in Figure 1. The lines with

```
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
```

Figure 1: Sample Makefile

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 *pre-requisites*, 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.

In the time 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 [5] for an analysis of the chief challenges to make). IDE's are often capable of maintaining dependencies automatically through an 'understanding' of the semantics of the systems they integrate. For example, Microsoft's Visual C++ IDE maintains its own make-like description file, which is not directly modified by the programmer. 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

[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.

SIGSOFT '96 CA, USA
© 1996 ACM 0-89791-797-9/96/0010...$3.50

production tools a project requires: the 'openness' of a tool like make enables the interoperation of a wide spectrum of tools possibly combined in original and unexpected ways, but this may not be possible for an IDE. It is therefore resonable to look for a basic theory of dependencies between software configuration items. This may lead to good approaches to communication between software environments combining the advantages of open systems like make with those of IDE's that address make's limitations.

This paper examines the idea of using an adaptation of a modelling technique from concurrent and distributed systems called a *Petri net* to model dependencies between software configuration items. The focused goal is to show how one can model and extend various approaches to optimizing builds of configurations. A concurrency formalism has been chosen because the typical state of a build configuration contains a great deal of potential course-grained parallelism, so it is natural to model the build directly as a concurrent computation. The goal here is not to provide a semantics of make, although make inspires a number of the problems considered, nor is it feasible in this short paper to deal with all of the issues that challenge a software configuration maintenance tool (for example, version control is not discussed). Instead, the aim is to treat rigorously the correctness criteria for build optimizations and the modular description of these optimizations, as, for instance, a collection of interoperating IDE's might report them in a suitable data structure.

The structure of the paper is as follows. After this introduction, the second section describes a structure called a *production net (p-net)* which is used to describe dependencies. The third section 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 fourth section introduces a concept of model for p-nets that enables the build-optimizations to be suitably modelled and their correctness criteria formalized and proved. The final section reflects on what has been accomplished and what more would be required to carry the formalisms further toward direct application. Since this is an abstract, the reader is referred to the full paper for proofs, further examples, and a fuller discussion of various computational and implementation-related issues.

2 Production Nets

In this section the graph structures used to represent relationships between software configuration items are introduced. They are a kind of Petri net, so the terminology and notation draws on that used for Petri nets. 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 *conditions,*
- E is a finite set of *events,*
- $S \subseteq B \times E$ is the *pre-condition* relation, and
- $T \subseteq E \times B$ is the *post-condition* relation. □

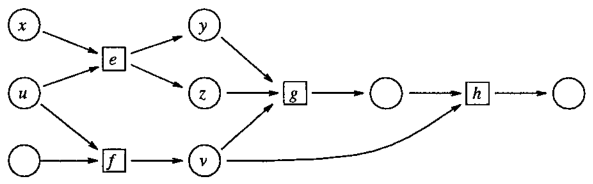


Figure 2: Example of a Net.

The letters x, y, z and the like are used to denote both conditions in B and events in E, whereas letters e, f, g are typically 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; for example $N' = (B', E', S', T')$. 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 *pre-conditions* and *post-conditions* of event e. Figure 2 shows an example of a net using the Petri net figure conventions: circles are used for conditions, 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 pre- and post-conditions of an event, so we define ${}^\bullet e^\bullet = {}^\bullet e \cup e^\bullet$.

Intuitively a software configuration is modelled as a net by treating software *items*—like main.o and main.c in the description file in Figure 1—as net *conditions*. On the other hand, *operations*—like 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 (conditions) within parentheses, which are reminiscent of circles, and the names of operations (events) within brackets, which are reminiscent of rectangles. The make description file of Figure 1 is depicted by the net in Figure 3. A formal treatment of the relationship between production nets and computations like those engendered by make description files are provided by p-net models, which are the topic of Section 4 below.

To provide a tractable theory for our purposes here, nets must satisfy certain axioms. We need some terminology to express the desired properties. Let N be a net. The *directed graph defined by N* is the directed graph which 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, 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 non-empty.

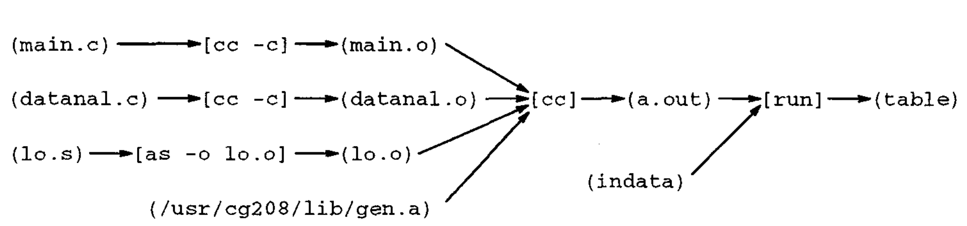


Figure 3: Net Corresponding to Makefile in Figure 1

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. Net (a) contains a cycle; net (b) fails to satisfy the unique-producer condition 2; nets (c), (d), and (e) have an event with pre-conditions or post-conditions that are unacceptable for condition 3; and net (f) has an isolated condition, in violation of condition 4.

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 conditions x such that there is a (unique) event e such that $e\ T\ x$; in this case we say that x is a *produced* item or condition 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, they are the ones that are modified by the environment (*viz.* programmers). Given a condition x, if there is no event e such that $e\ T\ x$, then x is said to be a *source* item (or condition). In Figure 2, x and u are source items, while y, z, and v are produced items. Items y and z are produced by event e, while item v is produced by event f.

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 conditions 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 conditions y, z are incomparable (with respect to $\sqsubseteq_N$ and $\sqsubseteq_B$) and events e, f are incomparable (with respect to $\sqsubseteq_N$ and $\sqsubseteq_E$). It is variously convenient to think of a software configuration in terms of these three orderings on its production net. The make description file essentially uses the colon notation to define the relation $\sqsubseteq_B$ on items (files). However, the evaluation of a make file is based on events (actions) structured by the dependencies in $\sqsubseteq_E$.

For any poset, $(P, \sqsubseteq)$, a subset $L \subseteq P$ is *left-closed* (relative to $\sqsubseteq$) if $x \in L$ and $y \sqsubseteq x$ implies $y \in L$. The notion of state for computation on a production net N will be modelled by left-closed subsets of $\sqsubseteq_N$. Given a subset $X \subseteq P$, the *left-closure* of X is the set

$$\downarrow X = \{y \in P \mid \exists x \in X.\ y \sqsubseteq x\}.$$

Computational state will be represented using certain special subsets of p-nets.

Definition: (Markings.) Let $N = (B, E, S, T)$ be a production net. A subset $M \subseteq B \cup E$ is *condition-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 condition-closed. □

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

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 pre-conditions of e are satisfied, 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 post-conditions are within M. A pictorial representation of the different states appears in Figure 5. Under suitable assumptions, the movement from one state to another with respect to e retains the property of being a marking.

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 $e \searrow M'$.* □

3 Computation Over Production Nets

The aim of this section is to provide motivation for the way in which computation over p-nets will be represented. The idea is similar to providing a set of operational rules for computation over a Petri net. However, the notion of marking used for production nets is somewhat different from that used for Petri nets. This difference is owing to the particular application for which p-nets are intended, which uses markings to model system build states. 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 pre-condition remains intact through the remainder of the build; sharing in the sense of these examples does not introduce conflict in the computation. So, the usual Petri net semantics of placing markings on conditions and having them consumed by events to which they are pre-conditions is not a convenient way of thinking of system

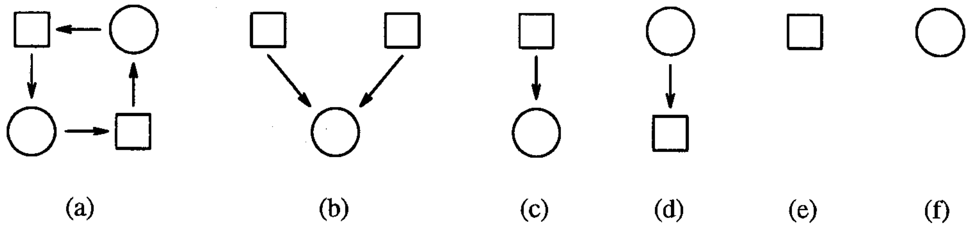


Figure 4: Nets that are not Production Nets

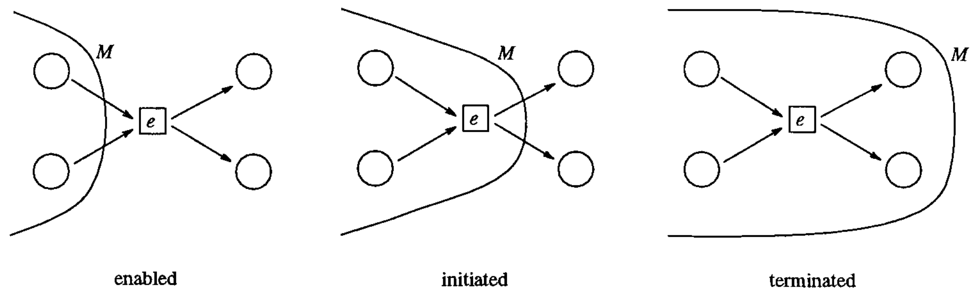


Figure 5: Three Relationships between Marking M and Event e

state for builds. Instead, one wants to view a build as having achieved consistency between sources and targets in a subset of the p-net that is left-closed relative to $\sqsubseteq_B$.

The pair of operational rules in Figure 6 provide a basic representation of the observable events of the concurrent computation. An event e engenders two forms of observable

Initiation $$\frac{M/e}{M \xrightarrow{\hat{e}} M \cup \{e\}}$$

Termination $$\frac{e/M}{M \xrightarrow{\breve{e}} M \cup e^{\bullet}}$$

Figure 6: Basic Operational Rules

behavior: $\hat{e}$ is the initiation of e and $\breve{e}$ is its termination. For example, the net in Figure 2 has the following possible evaluation sequences starting from the marking consisting of its minimal three conditions to the marking consisting of the whole net:

$$\hat{e}\breve{e}\hat{f}\breve{f}\hat{g}\breve{g}\hat{h}\breve{h},\ \hat{e}\hat{f}\breve{e}\breve{f}\hat{g}\breve{g}\hat{h}\breve{h},\ \hat{e}\hat{f}\breve{f}\breve{e}\hat{g}\breve{g}\hat{h}\breve{h},$$
$$\hat{f}\hat{e}\breve{f}\breve{e}\hat{g}\breve{g}\hat{h}\breve{h},\ \hat{f}\hat{e}\breve{e}\breve{f}\hat{g}\breve{g}\hat{h}\breve{h},\ \hat{f}\breve{f}\hat{e}\breve{e}\hat{g}\breve{g}\hat{h}\breve{h}$$

Concurrency of events is represented by the overlapping of their interval of execution, that is, the interval between $\hat{e}$ and $\breve{e}$ for any event e. A sequential computation over this net—such as most make evaluators provide—would have the form of either the first or last of these possibilities, in which no event begins before all the events that began before it have terminated.

Let us write $M \longrightarrow^* M'$ for the transitive, reflexive closure of the labeled relation. That is, $M \longrightarrow^* M'$ provided M is M' or there is a marking M'' and a label l such that $M \longrightarrow^* M'' \xrightarrow{l} M'$. The *number of steps* in the relation $M \longrightarrow^* M'$ is the minimum number of markings $M_0, \ldots, M_n$ such that $M_0 = M$ and $M_n = M'$ and there are labels $l_1, \ldots, l_{n-1}$ such that the relations $M_0 \xrightarrow{l_1} \cdots \xrightarrow{l_1} M_n$ all hold. Although we will quickly replace the simple semantics provided by the rules in Figure 6 by something more realistic (and interesting), it is worth noting briefly the following basic property:

Proposition 2 *If M is a marking of a net N and $M \longrightarrow^* M'$, then M' is also a marking.* $\square$

The proposition is proved by an induction on the number of steps in the relation $\longrightarrow^*$. Each case follows immediately from Lemma 1.

A build over a p-net $N = (B, E, S, T)$ may now be viewed as follows. First, a selection $X \subseteq B$ of targets is made. Then an initial marking M is chosen to consist of the minimal elements in $\downarrow X$. From this initial marking, the events in $\downarrow X$ are executed to produce a marking M' which contains X. The events may evaluate as concurrently as their dependencies allow until all events in $\downarrow X$ are terminated in M'. However, this is potentially a very inefficient way to ensure that targets properly reflect changes. For example, it may be that no condition among the minimal elements of $\downarrow X$ has been modified since the last time the targets in X were built, so *no* computation is required: the targets remain acceptable. The program make optimizes by examining the *dates* assigned to files by the operating system. Let us attempt to formalize this idea.

The dates associated with files by the operating system can be viewed as a function on conditions. An augmented form of production net includes the needed additional structure.

Definition: (DP-Nets.) A *dated* production net (dp-net) is a 5-tuple $N = (B, E, S, T, \delta)$ where (B, E, S, T) is a p-net and δ is a function from B into ω called the *date*. □

Given a dp-net $N = (B, E, S, T, \delta)$, it is convenient to define a pair of functions that provide greatest and least dates on pre- and post-conditions of events. The *pre-date* function ${}^\bullet\delta : E \to \omega$ is defined so that ${}^\bullet\delta(e)$ is the **greatest** value in the set $\{\delta(b) \mid b \in {}^\bullet e\}$. The *post-date* function $\delta^\bullet : E \to \omega$ is defined so that ${}^\bullet\delta(e)$ is the **least** value in the set $\{\delta(b) \mid b \in {}^\bullet e\}$. With this, the concept of an 'up-to-date' target is given as follows:

Definition: (Up-to-Date Markings.) A marking on the dp-net N is a marking on the underlying p-net (B, E, S, T). Such a marking M is said to be *up-to-date* if every event e whose post-conditions are contained in M satisfies ${}^\bullet\delta(e) < \delta^\bullet(e)$. □

A semi-formal explanation of the make 'date optimization' can now be given in terms of dated production nets. First of all, if M/e and the post-conditions of e are not up-to-date then the post-conditions of e must be built (initiation):

$$M, \delta \xrightarrow{\hat{e}} M \cup \{e\}, \delta$$

If, on the other hand, e/M holds, then the date function δ_1 is altered to a new date function δ_2 (termination):

$$M, \delta_1 \xrightarrow{\check{e}} M \cup e^\bullet, \delta_2$$

where δ_2 assigns dates to the postconditions of e that are later than the dates $\delta_1(b)$ for any b, and otherwise assigns the same dates as δ_1. Finally, if M/e but e is up-to-date, then no build is needed:

$$M, \delta \xrightarrow{\epsilon} M \cup \{e\} \cup e^\bullet, \delta$$

This rule, which captures the optimization of recognizing that an up-to-date event does not need to be run, is what we will call an *omission* rule. That this is correct is based on assumptions that are essential to the correctness of make-controlled builds: if a target exists, its pre-requisites exist, and the target is up-to-date with respect to its targets, then the target was created from the pre-requisites, and it does not need to be rebuilt from its pre-requisites if its pre-requisites are themselves based on up-to-date productions. We will return to the issue of build invariants and correctness later.

There are several ways in which the make optimization based on dates provides less than one would want in certain cases. Over the years, make extensions have attempted to address some of these problems, others are addressed only in IDE's. Let us consider three of these, each of which is based on common experience with system maintenance. The first concerns parsing tools, the second concerns SML programming language compilations, and the third concerns C header files.

Figure 7 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 from 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

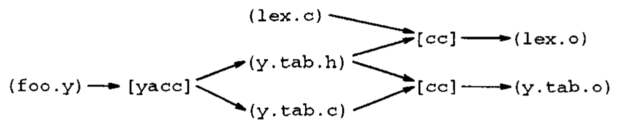


Figure 7: Identity of Old and New Inputs

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 in `foo.y` have been modified, then the generated 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 Figure 8. Unlike typical C pro-

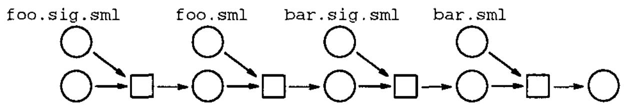


Figure 8: Hash Keys to Avoid Cascading Recompilation

grams, 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 generate 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 Matthias Blume [1], which is part of SML/NJ system, targets of compilations are assigned 'fingerprints'. A fingerprint is a bit string 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 9. Header files allow separate compilation of C

[2]The method for calculating fingerprints used in CM is based on

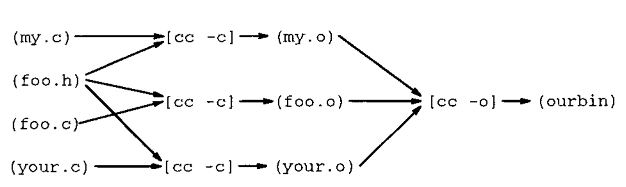
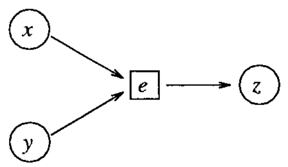


Figure 9: Changes in Headers

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 files are required. This has the advantage that one may compile C_i even if the C implementations of header files needed for the compilation are not available, perhaps because they are under development. In Figure 9 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 proto-type 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 [15] 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 9 should be taken with an appropriate grain of salt, but the basic idea is fairly language independent. The idea has inspired several subsequent studies, including one on 'smart*er*' recompilation [13] for C programs and 'smart*est*' recompilation for SML [14].

4 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 following very basic p-net:



This net has many models. For example, it might be 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. Similarly, the concept of a dated p-net is one in which the intended model is expected to associate dates with conditions. 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 [3], an abstract interpretation is based on the use of a 'non-standard' 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 reqired. 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. In examples, it will be convenient to write down some partial sections: 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.

A model of a p-net is a family of sets indexed over a subset of the conditions of the net and an 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

- a family of sets $V = (V_x \mid x \in B')$ indexed by conditions $B' \subseteq B$, 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, components 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 10, which corresponds

Rabin's CRC polynomials; see [2] for an exposition.

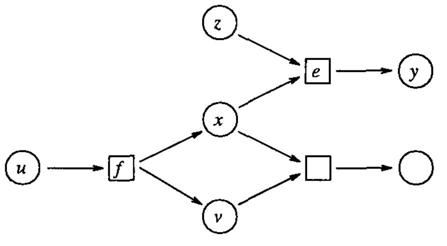


Figure 10: P-net for Example of a Model

to the one in Figure 7. The 'standard' model for the conditions 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, V_v, V_z are both the set of C files, and V_y is the set of object files. The events e, f are interpreted as follows: R_e is the relation between yacc input files and output files, R_f is the relation between C input files and the object files producted by compiling them.[3]

Given a model, the association of conditions and events to elements of the model provides the concept of system state:

Definition: (States and Consistent Events.) Suppose

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

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 $\mathcal{A}, s \models e$ just in case

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

An event e is said to be *consistent* in state s if $\mathcal{A}, s \models e$. A left-closed subset L of $B \cup E$ is consistent with respect to s if, for each non-minimal condition 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 a being a multi-sorted model of first-order logic that interprets relation symbols R_e corresponding to events $e \in E'$. Variables corresponding to conditions are interpreted in $\mathcal{A}$ by the state s. For example, for the net in Figure 2, $\mathcal{A}, s \models e$ is shorthand for something like $\mathcal{A}, s \models R_e(x, y, z)$. 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 $\mathcal{A}, s \models e$ is that it is dependent only on the values on ${}^\bullet e^\bullet$. 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 and f is undefined outside of U, but for the purposes of this paper it is taken to be the latter. The proof of the following can be obtained by simply unrolling the definition:

Lemma 3 *Let $\mathcal{A}$ be a p-net model N and let e be an event in N. For any pair of states s, s', if $\mathcal{A}, s' \models e$ and $(s \mid {}^\bullet e^\bullet) = (s' \mid {}^\bullet e^\bullet)$, then $\mathcal{A}, s \models e$.* □

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 conditions are interpreted as things like C source files and object files, while the model $\mathcal{B}$ instead interprets these conditions 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 $\mathcal{A}$ and $\mathcal{B}$ are models of the p-net $N = (B, E, S, T)$. 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 $U \subseteq B$:

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

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

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 have the desired relationship. 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 produced files may be deleted *but not modified.* Modifying a produced 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 produced 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 produced files, for example, to save

[3]A technical quibble here is that each of the sets V_i in this example is really the set of *files*. 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 produced files, which must be in the specific class of files that could be produced by programs like the C compiler.

space. 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: perhaps it is reasonable just to think of deletion as an extreme form of modification! 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 the desired abstractions. 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 [15] 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. Changes come in three forms: a source item is modified by the environment, an item is produced in the course of a build, or an item 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 apply to a set U of source items, or to a set $e^\bullet$ of produced items. We need a name for this pair of cases:

Definition: Let $N = (B, E, S, T)$ be a p-net. A *generation* of conditions 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. It will be useful for us to consider the restriction of f to the *complement* of the set U. This will be denoted $f \nmid U$.

Definition: Suppose $\mathcal{A}$ and $\mathcal{B}$ are models of the p-net $N = (B, E, S, T)$ and $\Phi : \mathcal{A} \rightarrow \mathcal{B}$ 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 $s \nmid e^\bullet = s' \nmid e^\bullet$ and $\mathcal{A}, s' \models e$.

An *abstraction server for* Φ is a function ϕ which takes as its arguments a generation of conditions, 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', generation $U \subseteq B$, and $\mathcal{B}$-state t:

$$\text{[Abstraction]} \quad \frac{\begin{array}{c} s \nmid U = s' \nmid U \\ \downarrow U \text{ is consistent in } s' \\ \Phi(s, t) \end{array}}{\Phi(s', \phi(U, s', t))}.$$

□

With these definitions it is possible to describe the rules for computation in Figure 11. These rules can be viewed as the generalization of the 'date optimization' rules described earlier to a class of similar optimizations determined by the choice of the abstraction $\Phi : \mathcal{A} \rightarrow \mathcal{B}$. *Initiation* occurs when a build must be carried out because the abstraction $\mathcal{B}$ does

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

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

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

Figure 11: Computation Relative to Servers α and ϕ

not indicate it is unnecessary. 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.

The principle soundness result for the rules in Figure 11 is the following:

Theorem 4 *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 t is a state of $\mathcal{B}$ such that $\Phi(s, t)$. Let M be a consistent marking of $\mathcal{A}, s$. If $M, s, t \longrightarrow^* M', s', t'$ 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 theorem is proved by inducting on the length of the evaluation from M, s, t to M', s', t'.

5 Applications of Abstractions

An application of the concept of an abstraction requires the 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:

Abstraction Model We must define a model $\mathcal{B}$ for N which is to serve as the space of abstractions. This entails selecting the conditions $B^{\mathcal{B}}$ 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 pre- and post-conditions 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.*

Date Abstraction

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 iff t is defined on ${}^\bullet e^\bullet$ and

$$\max\{t_x \mid x \in {}^\bullet e\} < \min\{t_y \mid y \in e^\bullet\} \qquad (1)$$

Relation The abstraction relation is defined by stipulating that $\Phi(s,t)$ holds iff, 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 generation of conditions and s' is an $\mathcal{A}$-state and 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

$$t'_x = 1 + \max\{t_x \mid x \in \downarrow\{x\} \cup \uparrow\{x\}\}. \qquad (2)$$

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:

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

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 Equation 1 holds. By the definition of $\Phi(s,t)$ these conditions imply $\mathcal{A}, s \models e$. 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 values 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 $\max\{t'_x \mid x \in {}^\bullet e\} < \min\{t'_y \mid y \in e^\bullet\}$ then Inequality 1 holds too, so $\mathcal{A}, s' \models e$ follows from $\Phi(s,t)$ and the fact that s' has the same values as s on ${}^\bullet e^\bullet$.

[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.

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

$$\frac{\Phi(s,t) \qquad s \nmid U = s' \nmid U \qquad \downarrow U \text{ is consistent in } s'}{\Phi(s', t')}.$$

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

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

We must show that s' is defined on ${}^\bullet e^\bullet$ and $\mathcal{A}, s' \models e$. 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 generation 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. Suppose second 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 doesn't 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.

A Customized Abstraction

Let us consider the optimization proposed for the p-net in Figure 7. 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 10. Let $\mathcal{A}$ be the *standard* model for this production net as we described it earlier. The abstraction is as follows:

Model Only conditions 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 $V_y^{\mathcal{B}} = V_x^{\mathcal{A}} \times \omega$ and $V_z^{\mathcal{B}} = \omega$. The relation $\mathcal{B}, t \models e$ holds iff 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 iff 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 generation. 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 $\mathcal{B}, t \models e$. 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 difference abstraction, so s is defined on ${}^\bullet e^\bullet$ and $(x, z \mapsto u, s_z)\ R_e^A\ (y \mapsto s_y)$. But t also satisfies the first condition for the abstraction Φ, so $s_x = t_x = u$, which means $\mathcal{A}, s \models e$ as desired. Soundness with respect to deletion is straight-forward. 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.

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 projection onto the first coordinate, and $\mathrm{snd} : X \times Y \to Y$ be 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

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

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

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 condition $y \in e^\bullet$ 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 iff

- for every source x, $t_x \simeq s_x$, and
- for every produced x, t_x is defined iff 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 $\mathcal{A}, s' \models e$.

Server For any generation $U \subseteq B$ and $\mathcal{A}$-state s' and $\mathcal{B}$-state t define $\phi(U, s', t)_x \simeq s'_x$ if x is a source condition in U, but if x is a produced condition 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 $\mathcal{B}, t \models e$ for some s, t, e. We must show that $\mathcal{A}, s \models e$ also holds. The fact that $\mathcal{B}, t \models e$ means that there is a post-condition $y \in e^\bullet$ 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 $\mathcal{A}, s' \models e$. But, by Lemma 3, these facts imply that $\mathcal{A}, s \models e$, as desired. That Φ is also sound with respect to deletion is straight-forward, 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 conditions U.

To see that ϕ is a server for Φ, let U be a generation of conditions, let s be a $\mathcal{A}$-state, and let t be a $\mathcal{B}$-state. Suppose that $\Phi(s,t)$ and s' is a $\mathcal{A}$-state such that $s \nmid U = s' \nmid U$ 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, $\mathcal{A}, s' \models e$.

Fingerprinting Abstraction

The difference abstraction is inefficient in some ways: the abstraction keeps the entirity 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 and applied in numerous other contexts. To fit fingerprinting into the theoretical framework of this paper 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 supose 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 conditions 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 $\tilde{\mathcal{A}}$ iff the values of the pre- and post-conditions 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 $\tilde{\mathcal{A}}$ are isomorphic. Thus the fidelity of $\tilde{\mathcal{A}}$ to $\mathcal{A}$ is measured by how closely the fingerprinting function approximates being an injection. A server for $\tilde{\mathcal{A}}$ is also needed. For any e, s, x, define $\tilde{\alpha}(e,s)_x = (\alpha(e,s)_x, f_x(\alpha(e,s)_x))$.

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

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

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

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

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 $\tilde{\mathcal{A}}$-state s and $\mathcal{B}$ state t, the relation $\Phi(s,t)$ holds iff

- for every source x, $t_x \simeq f_x(s_x)$, and
- for every produced x, t_x is defined iff 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 $\mathcal{A}, s' \models e$.

Server For any generation $U \subseteq B$ and $\tilde{\mathcal{A}}$-state s' and $\mathcal{B}$-state t define $\phi(U,s',t)_x \simeq \mathrm{snd}(s'_x)$ if x is a source in U. If x is a produced condition 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).

6 Conclusions

The accomplishments of this paper are the introduction of production nets and their models, the formulation of abstractions and their associated correctness conditions, and the application of these concepts in a collection of noteworthy cases. Questions still remain about the integration of models, the way in which models should be described and implemented, and whether a sufficient range of problems that arise in real system configurations can be treated reasonably using the abstract framework described here. A more limited objective is applying the theory to more of the cases within its current realm; challenges include a rigorous treatment of abstractions like Tichy's smart recompilation and the treatment of systems with multi-pass builds (or apparent cycles) such as the type-setting program LaTeX. Let me close by commenting very briefly on the three larger questions.

From a mathematical perspective, the full version of this paper provides a satisfactory account of how various models and abstractions can be combined. To give a hint about the systems perspective of this mathematics, suppose we are given a pair of IDE's and a project that needs to manipulate items produced by them. Each of the IDE's controls the dependencies, abstractions, and build operations for a portion of the collection of items included in the project. Figure 12 illustrates the general idea. The IDE's supply the

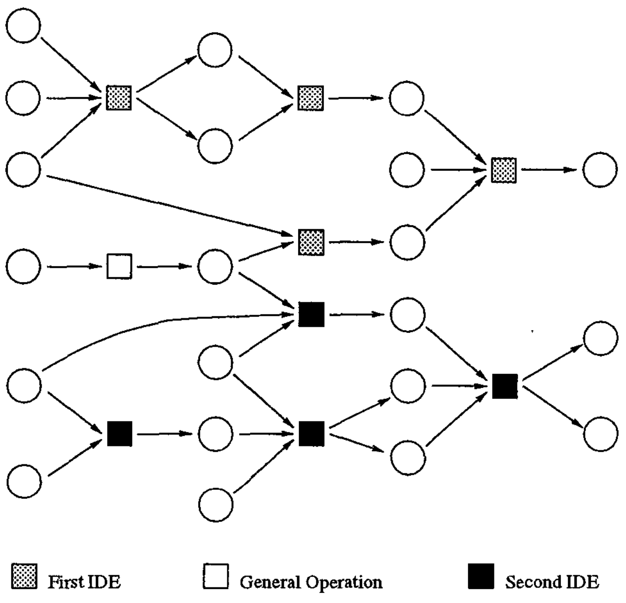


Figure 12: Integrating Models

servers that are used in the overall build, which is controlled by computation over the underlying p-net according to the rules in Figure 11. Abstractions may be supplied by the IDE's (based on special 'knowledge' they have about the semantics of the items in their domains) or by other means (as, for instance, `make` supplies the date abstraction or CM supplies the fingerprinting abstraction).

P-nets are, of course, a mathematical abstraction; a system that uses them must represent them in a data structure or allow the programmer (or a system) to describe them in a language. A tool like `makedepend` performs this latter function for C-programming items and `make`. Also `nmake` [5] provides explicit support for dependency-reporting. A more sophisticated and abstract language than that of `make` description files is provided by the *Vesta* programming language [7, 6] which is the description language for the Vesta configuration management system [8]. Vesta makes some decisions differently from the way p-nets are applied in this paper, but the approaches may be complementary in some ways. The full version of this paper describes implementations of some of the build computations using the language

Pict [11, 10, 12], which is based on Milner's π-calculus and provides useful high-level constructs for describing the concurrent computations over p-nets. Comparison with other configuration management languages and systems such as the various forms of concurrent `make` would be helpful. In any case, a suitable data structure or language for p-net models and abstractions requires further exploration.

As for whether a sufficient range of problems that arise in real system configurations can be treated reasonably using p-net abstractions and models, there are several issues that must be treated seriously before any attempt at validation seems worthwhile. Key issues include the treatment of versions/variants (a topic treated in other models such as Inscape [9] and feature logic [16]) and the incorporation of changes in dependencies (that is, where a change in a source item results in a change in the underlying p-net of dependencies). *Dynamic* determination of dependencies may also be worthy of consideration. Another interesting issue is the possibility of restricting one's view of a p-net of items by moving a *baseline* to hide or expose items to change control.

Acknowledgements

I would like to express my appreciation to the following people who influenced this work: Benli Pierce, Sandip Biswas, Luca Cardelli, Tony Hoare, Michael Jackson, Trevor Jim, Cliff Jones, Dave MacQueen, V. Mahesh, Andy Pitts, John Reppy, Glynn Winskel. 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).

References

[1] Matthias Blume. *CM: A Compilation Manager for SML/NJ.* User Manual.

[2] Andrei Broder. Some applications of Rabin's fingerprinting method. In R. M. Capocelli et. al., editor, *Sequences II: Methods in Communication, Security, and Computer Science.* Sprinter-Verlag, 1991.

[3] G. L. Burn, C. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[4] Stuart I. Feldman. Make—a program for maintaining computer programs. *Software—Practice and Experience*, 9:255–265, 1979.

[5] Glenn Fowler. A case for `make`. *Software—Practice and Experience*, 20(S1):S1/35–S1/46, 1990.

[6] Christine B. Hanna and Roy Levin. The Vesta language for configuration management. Technical Report 107, Digital Systems Research Center, 1993.

[7] Butler W. Lampson and Eric E. Schmidt. Practical use of a polymorphic applicative language. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, 1983.

[8] Roy Levin and Paul R. McJones. The Vesta approach to precise configuration of large software systems. Technical Report 105, Digital Systems Research Center, 1993.

[9] Dewayne E. Perry. Version control in the Inscape environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 142–149, Monterey, California, March 1987.

[10] Benjamin C. Pierce. Programming in the pi-calculus: An experiment in programming language design. Tutorial notes on the Pict language. Available electronically, 1995.

[11] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer-Verlag, 1995.

[12] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. To appear, 1995.

[13] Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, 1988.

[14] Zhong Shao and Andrew W. Appel. Smartest recompilation. In Susan L. Graham, editor, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 439–450. ACM, 1993.

[15] Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, 1986.

[16] Andreas Zeller. A unified version model for configuration management. In Gail Kaiser, editor, *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 20 (4) of *ACM Software Engineering Notes*, pages 151–160. ACM Press, October 1995.