Hierarchical Modularity: Compilation Management for Standard ML

Matthias Blume Andrew W. Appel
Princeton University

July 17, 1997

Abstract

Many programming languages identify modular units with compilation units, while only few extend this to permit hierarchies of language-level modules within individual compilation units. But when the number of such compilation units is large, then it becomes increasingly important that they themselves can be grouped together and that explicit export interfaces can be used to control namespaces. We explain how the group model implemented in SML/NJ's compilation manager CM provides the necessary facilities to avoid unwanted interferences between unrelated parts of large programs. We also discuss the problem of automatic dependency analysis as well as its complexity and show ways of avoiding intractability.

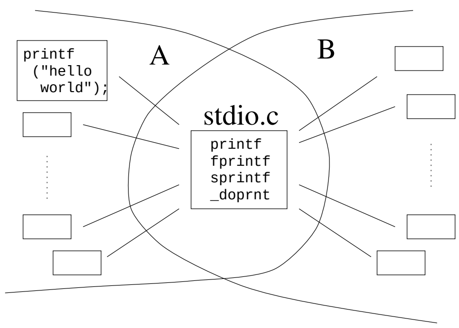

Figure 1: **Sharing one source file.** Only a few programming languages let the programmer control the interface of one source file to guarantee that in can be shared between several clients. In C one would declare `_doprnt` as *static*, because local functions should not be exposed. But the C programmer has no way to prohibit `stdio.c` from referring to symbols in system A; these references might prevent it from being used in system B.

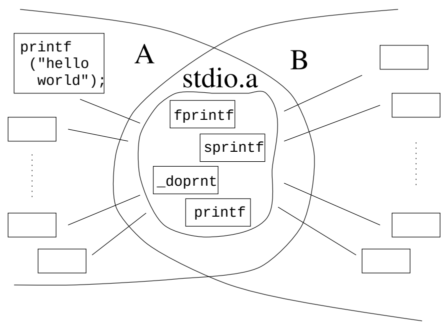

Figure 2: **Sharing groups of source files.** Often we want a group of sources to act like one module. An explicit interface placed on `stdio.a` as a whole (as opposed to its constituent components) guarantees the absence of undesired dependencies on either A or B and does not expose local objects like `_doprnt`. The C language has no mechanisms for expressing either of these requirements.

1 Introduction

Abstraction and modular design are key elements of modern software engineering. When the number of compilation units is large, then it becomes increasingly important that they themselves can be grouped together and that explicit export interfaces can be used to control namespaces. When there are many groups, then one may want to organize them into supergroups, and so on, thus forming a hierarchical modular structure.

We wish to structure big programs into grougs, subgroups, and libraries, the way operating systems structure their file system into directories with subdirectories. But we do not stop there, because—as we will show—groups and libraries can be powerful tools for increasing separation of unrelated parts of the program and for controlling interfaces between parts that are related. This will allow us to solve the problems shown in figures 1, 2, 5, and 6.

We define *compilation management* in such a system to comprise:

- calculating the dependency DAG between individual source files;
- determining which subset of sources need to be compiled or recompiled;
- determining the order of compilation;
- defining the relationship between source files and program libraries;
- controlling the visibility of globally defined names.

The last two items—involving the management of the namespace—are the important new topics addressed by our work; but name visibility affects dependency calculation, as we shall show.

Figure 1 shows a situation where one module (`stdio`) is shared by two different projects (A and B). We would like to administer its interface so that that the compiler can guarantee the absence of unwanted dependencies on either A or B. Even that is not fully supported by most programming languages. But for large-scale programming we need to take this further: It is important that `stdio` itself can be a group of source files, and that there can be a summary interface that controls such a group's interface to its various clients (see figure 2).

The global namespace, which is inhabited by "external" symbols, and which is commonly used by the program linker, poses a problem for modularity. Since there is only one such namespace, any given identifier can only have one definition at a time. Thus, there is a potential for conflicting definitions, which is especially troublesome when the interfering parts of the program are unaware of each other's existence, as is often the case with libraries.

Definitions are said to *interfere* if they cause restrictions on parts of the program that are neither directly nor indirectly related (figures 3 and 4). We call

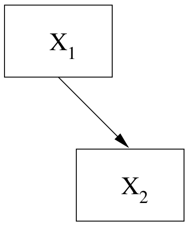

Figure 3: **Direct relation.** Two program fragments X_1 and X_2 are directly related if X_2 explicitly refers to a definition that is exported from X_1.

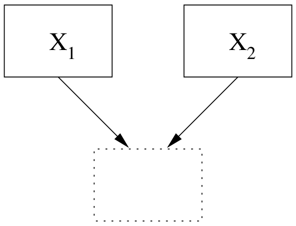

Figure 4: **Indirect relation.** Two program fragments X_1 and X_2 are indirectly related if there is a third fragment that explicitly imports from both.

restrictions *modular* if they only affect related parts of the program.

This work's primary concern is to present a framework that avoids all non-modular restrictions on *definability* of identifiers and *availability* of definitions. We will also show how the restrictions that are modular can be resolved by simple local modifications to only the part of the program where they occur.

Definability: A program in a language like C [KR88] cannot use certain identifiers, because they are potentially taken by libraries or other parts of the program, even if those are conceptually unrelated. Thus, these identifiers are not definable anymore (figure 5).

Availability: Programs in other languages, for example SML [MTHM97], can have arbitrarily many definitions for the same name, but although this eliminates restrictions on definability it creates new restrictions on availability: Parts of a larger program may not be able to see an early definition for variable x, because there is a different, intervening definition for x inhibiting access to

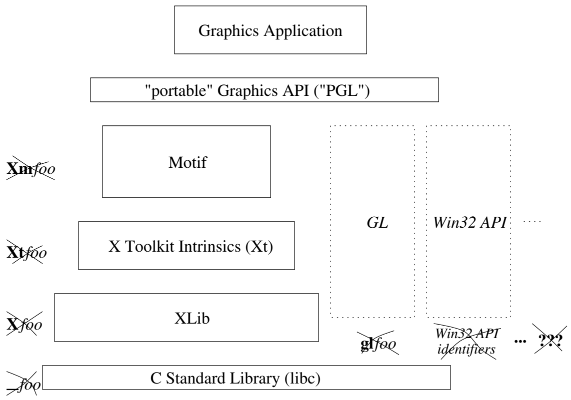

Figure 5: **Restrictions on definability inhibit modularity.** The programmer of this graphics application in C, who ideally would like to use the “portable” PGL library abstractly, as a black box, must be careful not to interfere with any of the various libraries upon which PGL might be implemented. Entire classes of identifiers must be avoided: **Xm***xxx*, **Xt***xxx*, and so forth because of X Windows, **gl***xxx* to avoid conflicts with GL, scores of symbols to account for Win32, but also many names that the creators of new library designs *might* choose for their purpose in the future.

the one that was intended (figure 6).

Because of the global namespace’s lack of structure, definitions that are conceptually local to a small group of sources are often promoted to be globally visible.

2 The compilation manager CM

CM—the compilation manager for SML/NJ—provides hierarchical modularity through its group model. The description of a group consists of three elements:

- a set of source files
- a set of imported groups
- a list of exported symbols.

The minimal design of CM’s configuration language (see the example in figure 7) makes it easy to use, especially because built-in automatic dependency analysis eliminates the need for being explicit about order of compilation within each group.

Sources of a group should conceptually belong together. They may correspond to a subcomponent of a

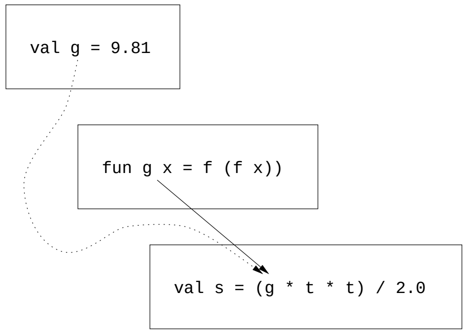

Figure 6: **Restrictions on availability inhibit modularity.** In this SML program a value for `s` was meant to be calculated in terms of the gravitational constant `g` that was defined earlier. But a third compilation unit that is conceptually unrelated redefines `g`, so the original definition becomes unavailable. In this case the resulting program does not even typecheck and the problem can be detected at compile time; in other cases the program might produce an unintended result.

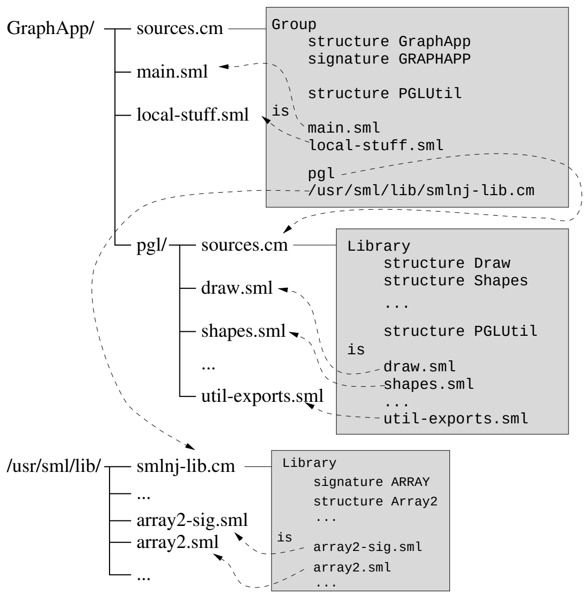

Figure 7: **CM group description files.** This figure shows a sketch of a program, its group descriptions, and a sample directory hierarchy to hold the associated files. Directory `GraphApp` contains source files (`main.sml`, `local-stuff.sml`) and the description (`sources.cm`). The application exports symbols `structure GraphApp`, `signature GRAPHAPP`, and `structure PGLUtil`. `PGLUtil` itself is imported from our graphics library “PGL” described by `pgl/sources.cm`. In addition to that there are imports from the SML/NJ library, which was installed in a central location by the system administrator. (Relative pathnames in description files refer to files in the directory that contains the description.)

larger program, or they may act as a program library. If group A needs to refer to definitions exported from another group B, then B must be mentioned in A's description.

Names exported by the modules of a group are visible in other modules of the same group, but only those that appear in the list of exported symbols can also be seen from the outside.

Without CM's groups the client of PGL in figure 5 could not use certain identifiers, because they are potentially taken by the libraries that implement PGL. To cope with this we implement PGL as a group whose export filter only lists those identifiers that are supposed to be accessible by its clients. Consequently, the application programmer cannot even tell how PGL is implemented; the application code will be truly independent of the libraries underlying PGL's concrete realization.

CM's group model is able to eliminate modularity-inhibiting restrictions on definability and availability in a general way. The following section will discuss this more formally.

3 Environments, linking, and hierarchical modularity

Cardelli [Car97] presents an excellent discussion of the problems that arise with modules and separate compilation. His notion of a linkset is used as a framework for describing and reasoning about consistent, type-safe linking. Type-safe linking, for example provided by SML/NJ's "visible compiler" [AM94], is a prerequisite—but not the focus—of our work. In place of Cardelli's linksets our notation uses functions to express operations on environments and equations for describing their properties. This also reflects the actual implementation of CM more closely.

Environments. During separate compilation individual sources are always compiled with respect to some environment that represents the definitions exported from other compilation units.

Formally, an environment $\rho \in U$ is a partial mapping from identifiers Ide to denotations D.

$$\rho \in U = \text{Ide} \rightarrow D$$

D depends on the programming language. In SML it would correspond to the compilation unit's static and dynamic semantics. For C, on the other hand, external identifiers stand for machine addresses. In this case D would be a domain of locations. To abstract from such language-dependent issues we will instead use a domain of labels $l \in \text{Lab}$. These labels uniquely identify each definition of a given program. Thus, we can always distinguish between different bindings for the same identifier without having to consider the meaning of an identifier according to the semantics of the language.

$$D = \text{Lab}$$

The domain $\mathit{dom}(\rho)$ of an environment ρ is the set of identifiers that are bound there. $\emptyset_U$ is the environment with an empty domain.

Environments can be combined using the $+$ operator:

$$\begin{aligned} + &\in U \times U \rightarrow U \\ \rho_1 + \rho_2 &= \lambda x. \begin{cases} \rho_1(x); & x \in \mathit{dom}(\rho_1) \\ \rho_2(x); & \text{otherwise} \end{cases} \end{aligned} \qquad (1)$$

The operator $+$ is associative, but is not commutative if $\mathit{dom}(\rho_1) \cap \mathit{dom}(\rho_2) \neq \emptyset$.

Compiling and linking. We use the *def* operator (see figure 8) as an abstraction of compiler and linker. It calculates an incremental delta environment containing just the bindings corresponding to definitions that are explicit in the compilation unit:

$$\mathit{def} \in \text{Source} \times U \rightarrow U$$

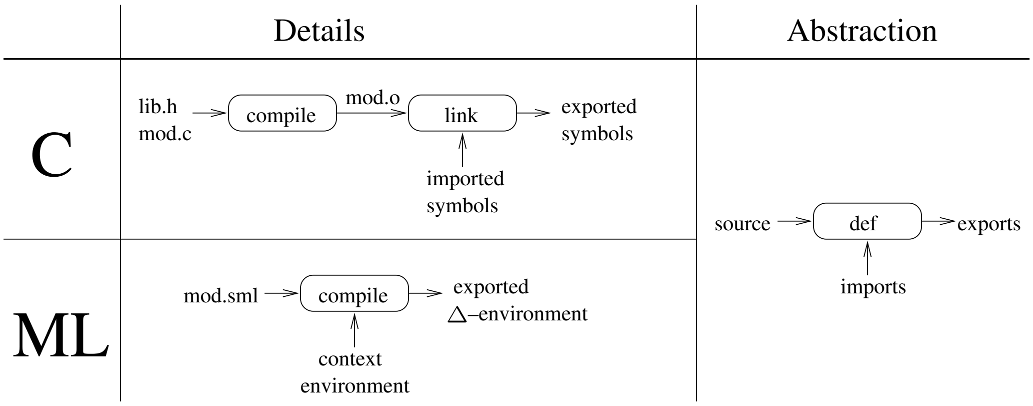

Figure 8: **Compiling, linking, context.** A C source file is first compiled and subsequently linked with respect to a table of imported definitions. In the case of SML compiling and linking are combined into one step, but, again, there is a context environment representing definitions that are imported from other compilation units. We abstract from language-specific differences and uniformely use the *def* operator as a model for compiling (and linking) of a source with respect to some context.

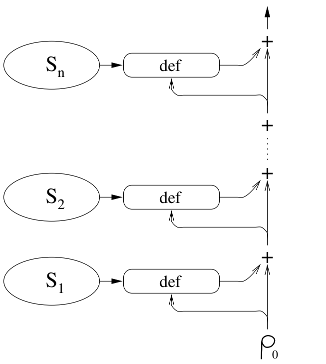

Figure 9: **Compilation environments for SML.** In SML the context environment that is used for compiling a source is built incrementally by layering the exports of sources that were compiled earlier on top of the initial basis environment ρ_0.

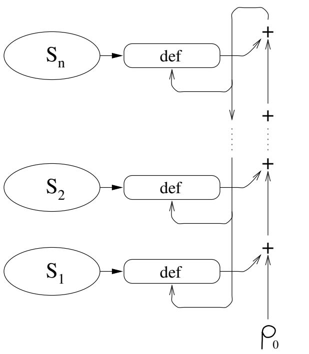

Figure 10: **Compilation environments for C.** Conceptually, every source of a C program is compiled with respect to the same global environment, which itself is constructed by layering the exports from all sources on top of some initial basis environment ρ_0. Implementations resolve the circularity at link-time; the compiler takes information from header files and uses it as an approximation of the global environment.

Programming languages differ in how they calculate the input environment for *def*. Consider a program consisting of sources $S_1, \ldots, S_n$ written in SML. The export environment ρ_i of source S_i is

$$\rho_i = \mathit{def}(S_i, \rho_{i-1} + \cdots + \rho_1 + \rho_0)$$

where ρ_0 is the initial basis environment. This situation is depicted in figure 9.

In C, on the other hand, every source file is linked in the context of the same environment. That global environment is constructed from all exports of all sources (see also figure 10):

$$\begin{aligned} \rho_i &= \mathit{def}(S_i, \rho); \qquad \forall i \in \{1, \ldots, n\} \\ \rho &= \rho_n + \cdots \rho_1 + \rho_0 \end{aligned}$$

The process of linking corresponds to solving the system of simultaneous equations. In C none of the identifiers can be bound by more than one environment ρ_i:

$$i \neq j \Rightarrow \mathit{dom}(\rho_i) \cap \mathit{dom}(\rho_j) = \emptyset$$

Because of the resulting commutativity of $+$, linking is order-independent and becomes straightforward. However, with each source the programmer must provide header files containing the information necessary to construct an incomplete version of ρ that is suitable for compiling the source, because the full ρ only becomes available after all sources have been compiled. This is sometimes rather cumbersome, but unlike SML it allows for mutual recursion across compilation units.

Linking subsets of sources. We define

$$\textit{multidef}_{\textit{lang}} \in 2^{\text{Source}} \times U \to U$$

to be the language-dependent extension of *def* to sets of sources.

C: The $\textit{multidef}_{\text{C}}$ operator passes its context argument to individual calls to *def* for each of the constituent sources. The resulting delta environments are combined using $+$, thus yielding a delta environment for the entire subset (see figure 11):

$$\textit{multidef}_{\text{C}}(\{S_1, \ldots, S_n\}, \rho) = \textit{def}(S_1, \rho) + \cdots + \textit{def}(S_n, \rho)$$

SML: $\textit{multidef}_{\text{SML}}$ must first use a dependency analyzer $\mathcal{A}$ (see section 4) to turn the set of sources into a sequence:

$$\mathcal{A} \in 2^{\text{Source}} \times U \to \text{Source}^*$$

The *step* function then incrementally builds the resulting export environment ρ'. Intermediate values for ρ' are layered on top of the initial context ρ to serve as the context needed for compiling individual sources of the set:

$$\textit{step}(\langle\rangle, \rho, \rho') = \rho'$$
$$\textit{step}(\langle S_i, S_{i+1}, \ldots\rangle, \rho, \rho') = \textit{step}(\langle S_{i+1}, \ldots\rangle, \rho, \textit{def}(S_i, \rho' + \rho) + \rho')$$

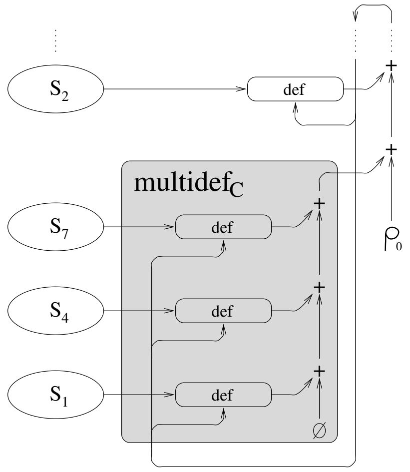

Figure 11: **Linking subsets of C sources.** The $\textit{multidef}_{\text{C}}$ operator is the C-specific extension of *def*, which calculates the export environment for a given subset of sources. Note that $\textit{multidef}_{\text{C}}$ returns an incremental delta environment that only binds symbols which are explicitly defined in that subset of sources. This is consistent with the way *def* works for single sources.

$$\textit{multidef}_{\text{SML}}(s, \rho) = \textit{step}(\mathcal{A}(s, \rho), \rho, \emptyset_U) \qquad (2)$$

Exports from each source are combined to eventually form the exports of the entire sequence; a second layering operation per source is necessary to form the corresponding compilation context (see figure 12). This explains why there are two occurences of $+$ in the definition for *step*.

3.1 Definability and availability

An identifier x is said to be *definable* in source S if it is legal for S to export a binding for x. The definition that binds identifier x at label l is *available* in environment ρ if $\rho(x) = l$.

Restrictions on definability or availability are not modular. The purpose of CM's group model is to curb undesired long-range effects of global definitions in order to eliminate such modularity-inhibiting restrictions.

Example: C. All external definitions of a C program are available everywhere, but an identifier x is definable in S_i only if it is not defined anywhere else:

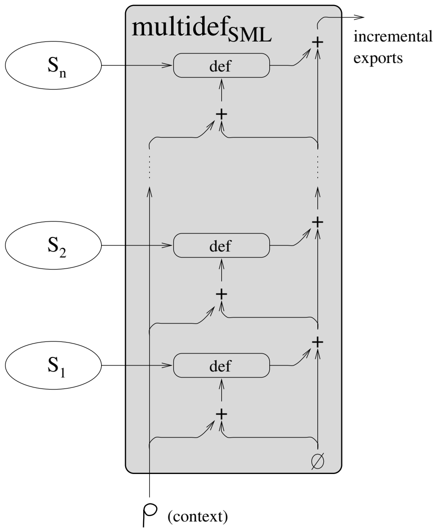

Figure 12: **Compiling (ordered) sets of SML sources.** The *multidef*$_{\text{SML}}$ operator extends *def* to sets of SML sources. Individual exports are incrementally layered on top of an initially empty environment. The result is a delta environment only containing bindings for identifiers defined in $\{S_1, \ldots, S_n\}$, which is analogous to the behavior of *def*.

$$x \text{ definable in } S_i \text{ iff}$$
$$\forall j : j \neq i \Rightarrow x \notin dom(\rho_j)$$

Thus, in C a definition in one source file affects definability and availability in the entire program. This creates an unfortunate implicit coupling between compilation units that is not modular, because it means that the programmer has to be aware of every identifier in every part of the entire program including all of the program libraries that it could be linked with.

Example: SML. The treatment of SML requires a well-defined ordering among the sources: S_1 will be compiled before S_2, and so on. Unlike in C every identifier is definable everywhere[1], but a definition in source S_i that binds x is available in S_j only if $i < j$ and there is no other definition for x in one of the sources "between" S_i and S_j:

$$\text{definition for } x \text{ from } S_i \text{ available in } S_j \text{ iff}$$
$$\forall k : (i < k) \wedge (k < j) \Rightarrow x \notin dom(\rho_k)$$

Here definitions do not interfere with the definability of names as in the case of C; instead, a definition for x makes previous definitions for x unavailable. The programmer has to be aware of all the definitions exported from *earlier* sources. Again, the resulting implicit coupling inhibits modularity.

Availability is sensitive to the order of compilation, which is especially troublesome when chosen automatically and not by the programmer. But problems only arise because a module can redefine symbols that are already defined. This not only makes the code difficult to read for a human—it also makes dependency analysis NP-hard (section 4).

3.2 Groups

It is only natural that the part of a program that sees variable x bound to l_1 cannot simultaneously see another definition that binds x to l_2. But at the time when the code mentioning x was written the programmer certainly had a specific idea about which binding x is supposed to refer to. To make sure the right thing happens we need a way to state our intentions explicitly, so compilation management can enforce them.

Our solution is to introduce *groups*, which are used to localize the effects of definitions. A group is meant to be a collection of sources that conceptually belong together. They may correspond to subcomponents of larger programs, or they may provide the functionality of program libraries.

If a source in group G_2 refers to a definition exported from another group G_1, then the description of G_2 must name G_1 explicitly as one of its imports. The resulting explicit export-import relation between groups can now be used to establish rules for removing unwanted interference between definitions in "unrelated" groups.

[1]This is a slight oversimplification, because the declaration `val y = x` does not provide a new definition for `y` if `y` is currently a constructor tag for some datatype.

Two groups are directly related if one explicitly imports from the other, they are indirectly related if a third group imports definitions from both, otherwise we call them "unrelated" (see also figures 3 and 4).

Formally, a group $G \in \text{Grp}$ is a triplet (s, i, e). Here, s is the set of sources, i is the set of imported groups, and e is a set of identifiers that is used for thinning the group's export interface.

$$\text{Grp} \subset 2^{\text{Source}} \times 2^{\text{Grp}} \times 2^{\text{Ide}}$$

No group may be in the transitive closure of its own imported groups. The graph of direct depencencies will therefore be acyclic.

Thinning can be understood as a filter operation $\mathcal{F}$ applied to an environment. The filter removes all bindings to identifiers that are not specified in the argument set; the filtered environment $\mathcal{F}(\rho, e)$ is ρ with its domain restricted to e:

$$\begin{aligned} \mathcal{F} &\in U \times 2^{\text{Ide}} \to U \\ \mathit{dom}(\mathcal{F}(\rho, e)) &= \mathit{dom}(\rho) \cap e \\ x \in e &\Rightarrow \mathcal{F}(\rho, e)(x) = \rho(x) \end{aligned}$$

Let ρ_0 be the initial basis. $\mathcal{C}(i)$ is the context environment built from the imported groups i, that is used when compiling the set of sources of a group (s, i, e):

$$\begin{aligned} \mathcal{C} &\in 2^{\text{Grp}} \to U \\ \mathcal{C}(i) &= (\sum_{I \in i} \mathcal{E}(I)) + \rho_0 \end{aligned} \tag{3}$$

$\mathcal{E}(G)$ is the export environment of group G:

$$\begin{aligned} \mathcal{E} &\in \text{Grp} \to U \\ \mathcal{E}(s, i, e) &= \mathcal{F}(\mathit{multidef}_{\text{SML}}(s, \mathcal{C}(i)) + \mathcal{C}(i), e) \end{aligned}$$

Notice, that a group (s, i, e) can re-export part of its own context $\mathcal{C}(i)$—provided it is named in e and not redefined by any of the sources $S \in s$.

We have chosen to consider unordered sets of imported groups. But the summation in equation 3 is order-independent only if the domains of the imported environments are disjoint. Therefore, we must require that that indirectly related groups cannot export definitions for the same identifier:

$$\begin{aligned} &\forall (s, i, e) \in \text{Grp} : (\{I_1, I_2\} \subset i \Rightarrow \\ &\qquad \mathit{dom}(\mathcal{E}(I_1)) \cap \mathit{dom}(\mathcal{E}(I_2)) = \emptyset \end{aligned} \tag{4}$$

3.3 How interferences can be resolved

An implementation of the group model immediately removes the vast majority of unwanted interferences between definitions in large programs, because definitions can only interfere when their respective environments are joined using $+$. With groups most environments never meet each other that way.

A very important property of our model is that any remaining interferences can always be resolved locally—by modifying only the group where they cause a problem. Three situations can arise:

1. The export environments for two sources in the same group contain definitions for the same identifier. These environments are joined using $+$ according to equation 2, which defines $\mathit{multidef}_{\text{SML}}$.
2. A source provides a definition for an identifier that is also defined by one of the imported groups.
3. In violation of equation 4 two imported groups independently provide definitions for the same identifier.

Interferences of the first kind can always be removed by locally changing one of the offending sources.

The second kind—a clash between a definition obtained from one of the imported groups and a definition in one of the sources—is legal in our model, and its behavior is given by equation 3: The definition from the group's sources overrides the one stemming from imported groups. Sometimes this is not what was intended, but the situation still cannot be resolved by a simple change to one of the group's sources. In such cases it becomes necessary to rename the identifier that is used to access a particular binding from an imported group at the point of import. This technique does not require changes to the exporting group, and it can also be used to resolve clashes of the third kind.

3.4 Groups vs. modules, namespaces, packages, ...

Some programming languages offer facilities to help fight the pollution of the global namespace. SML has structures and functors, Modula-3 [CDG$^+$89] has modules, C++ [ES90] has namespaces, and Java [AG96] offers packages.

The main difference in comparison to CM's groups is that these facilities are expressed in source language; modules, structures, namespaces, and packages have names themselves! Consequently, although less likely, there is always the potential for these names to cause the same kinds of problems they were supposed to solve by removing other names from the global namespace.

Of course, CM groups must also be named. The current implementation uses the name of the description file for this purpose. The crucial distinction is that the group name never appears in source code—only in group description files where they can be adjusted without causing recompilation. This way naming remains flexible. Groups can be moved, renamed, or replaced freely without having to touch the program's source code.

3.5 Realization in CM

The specification for a group must be written by the programmer. Group description files consist of two main parts: the list of exported identifiers and the list of members. Each member is either a source filename or the name of another description file, in which case the corresponding group will become an imported group.

There are two types of groups: "ordinary" groups and libraries. Ordinary groups do not require an explicit export list. If it is omitted, then CM will export all external identifiers with the exception of those coming from imported libraries.

As an important optimization CM always compares the context environment (equation 3) to the one used when previously compiling the same source (provided the source itself has not been modified). Recompilation can be avoided if source code and compilation context are unchanged. This strategy is known as *cutoff recompilation* [ATW94], because once a modification is found to have no effect on some source recompilation will not propagate any further from there.

4 Automatic dependency analysis

The problem of dependency analysis for SML is the following: For a given set of sources $s = \{S_1, \ldots, S_n\}$ and a context $\rho_0 \in U$ we are looking for a sequential arrangement $\vec{s} = S_{p_1}, \ldots, S_{p_n}$ of the same sources with the following property: If a variable x occurs free in S_{p_k}, then x must be defined "earlier":

$$\begin{aligned} x \;\in\; & \mathit{dom}(\rho_{k-1} + \cdots + \rho_0) \\ & \text{where } \rho_i = \mathit{def}(S_{p_i}, \rho_{i-1} + \cdots + \rho_0) \end{aligned} \qquad (5)$$

The sequence $\vec{s}$ implies a total order $\prec$ on s that we call *feasible*.

Thus, in a context ρ the dependency analyzer $\mathcal{A}$ forms a sequence $\vec{s}$ from a set of sources s:

$$\mathcal{A} \in 2^{\text{Source}} \times U \rightarrow \text{Source}^*$$

Equation 2 shows how dependency analysis is incorporated into our group model.

By equation 5 the definition for x that is referred to in S_{p_k} can be provided by either the context ρ_0 or by one of the other sources: $x \in \mathit{dom}(\rho_0) \vee \exists j : (j < k \wedge$

$x \in dom(def(S_{p_j}, \rho_{j-1} + \cdots + \rho_0)))$. However, this is not sufficient: Let x be a long identifier of the form $Y.z$, referring to a definition in structure Y. Another source S' could define structure Y without defining $Y.z$. But definitions for Y that do not define $Y.z$ make previous definitions for $Y.z$ unavailable.[2]

Once structures are permitted there are two aspects of SML that complicate dependency analysis:

1. symbols defined at top level in more than one source
2. the use of the **open** syntax at top level

We will see that either feature independently makes the problem intractable, and we will therefore address them separately.

4.1 Multiple definitions

Multiple definitions of the same symbol at top level in separate sources can introduce ambiguities into the association between uses of a variable and the corresponding definition.

Claim 1 *With multiple top-level definitions for the same identifier the problem of finding a feasible ordering is NP-complete.*

Proof. By reduction from 3-SAT.[3]□

We felt that it is reasonable to require having at most one definition for each top-level symbol per group. Although there are circumstances when one would want to override a given definition with a new one, CM addresses this issue adequately by introducing the notions of libraries and sub-groups. In this discussion of dependency analysis imported groups of a group are represented abstractly as part of the context environment, and top-level definitions for symbols that were already defined by the context are permitted in this model.

[2]The formalism in this summary version of the paper does not account for long identifiers, which, for example, would require adjusting equation 1.

[3]The complete proof can be found in [Blu97], and will appear in the full version of this paper if space permits.

Restriction 1 *In each group there can be at most one source that provides a top-level definition for any given symbol.*

We implemented this restriction in CM, and to our knowledge there has never been an instance where it caused difficulties to users. It provides a well-defined association of the symbols defined with the sources they are defined in. Therefore, dependency analysis can be performed by a depth-first search on the resulting use-definition graph, which can be done in time linear in the number of edges. Those edges correspond to the free occurences of symbols in SML sources.

Intuitively, $S_1 \prec S_2$ means that S_2 "depends" on S_1. If we want to avoid unnecessary recompilation we must capture the idea that two sources do *not* depend on each other. Total orders contain "too many" relations, but restriction 1 enables us to consider partial orders. For the sake of semantic predictability we desire a feasible partial order $\prec_{\min}$ that contains the fewest relations and is unique.[4]

4.2 Opening structures

SML programs that do not make use of the **open** syntax have a convenient property: Both the set of free variables of a source and the set of exported top-level definitions can be determined by scanning only the source itself; it is not necessary to know the definitions for any of the free variables.

The ability to open a structure, thereby making its constituent definitions directly available without need to use long identifiers, comes at the cost of losing this property. The problem is that **open** introduces a number of definitions, but the names so defined are not lexically apparent. In the scope of such a set of "indirect" definitions it may be that what looks like a free variable is actually bound, and what looks bound

[4]CM ignores the issue of link-time side effects during module initialization.

under superficial inspection may in certain cases actually be a free occurence. Here is an example of the latter:

```
structure S = struct
  val x = 1
end
open X
open S
val y = x + 1
```

Opening structure `S` seems to bind variable `x`, but if structure `X`—about which nothing is known—contains a sub-structure `S` without a variable `x`, then `x` is actually free in this code.

Language constructs that—like Pascal's **with** [JW78]—bind identifiers implicitly have been criticized before [Ten81, section 6.2.3.]. A feature that confuses dependency analysis tools will not be easy to understand by the human reader.

Ada's **use** [Ada80], **import** in Java, or **using** in C++ can behave like **open** in SML—with all the same implications for dependency analysis.

Modula-2 [Wir82] and Modula-3, on the other hand, do not have this problem. In Modula-2 one must write

```
FROM M IMPORT a, b, c;
```

in order for `a`, `b`, and `c`—and only those—to become directly accessible without being prefixed by `M`. Therefore, the identifiers defined are lexically apparent even without knowing the definition of module `M`. The design of Oberon takes this much further by discarding both **with** and **from-import**, leaving the language without any facility for circumventing the qualification of identifiers [Wir88].

In the context of dependency analysis it is especially troublesome that **open** at top level takes away the analyzer's ability to determine the set of exported definitions by simply scanning the source code. Instead, it will have to process **open** as it goes, which is complicated by the fact that in general yet-to-be-determined knowledge about the dependencies would be required for this. Even after banning multiple definitions for the same name dependency analysis is NP-complete if the use of **open** is not restricted.

Claim 2 *Dependency analysis is NP-complete for programs with* **open** *where multiple definitions for top-level names are prohibited, but where a top-level definition can override a binding in the context.*

Proof. By reduction from SAT.[5]□

If we ban **open** from the top level, then the definitions exported by a source can be determined by looking at just that source. Thus, whenever the dependency analyzer has to process an internal **open** it will know already where to find the definition of the structure that is being opened. The problem is tractable again.

Restriction 2a *The* **open** *syntax cannot be used at top level.*

In the current implementation of CM we enforce this rule. We believe in a programming style that uses SML's module language extensively, so there would be no need for **open** at top level. However, in some instances such a complete ban was prohibitive. For example, someone who is using Concurrent ML [Rep91] extensively—as a programming language in its own right—might want to open the `CML` structure to have more convenient access to its components.

There are several ways of restricting the use of top-level **open** in a more relaxed way. Here is one example:

Restriction 2b *Instances of the* **open** *syntax at top level are not permitted to introduce definitions for structure names that are used somewhere within the*

[5] For the complete proof see [Blu97] or—space permitting—the full version of this paper.

group if the context already provides a definition for them.

The more permissive restriction 2b no longer provides a way of calculating the set of exported definitions by processing a source in isolation from the other sources. However, a more involved algorithm that essentially processes all sources concurrently is still capable of finding the dependency graph and proving it unique in polynomial time.

A version of such an algorithm had been implemented in SC, which was CM's precursor [HLPR94]. But the authors of SC were not aware of the general problem's complexity class, so they only enforced restriction 1 and made no attempt of restricting the use of top-level **open**. As a result the analysis performed by SC was unsound in the sense that for certain programs it would fail to find an existing feasible ordering. Furthermore, for some programs with ambiguous dependencies it would silently pick one of the choices without warning the user about the existence of others that differ semantically.

4.3 Optimizations

It is relatively expensive to parse an entire source file every time the dependency analyzer runs. CM therefore calculates a condensed version of the source, which sheds all parts of the abstract syntax tree that are not necessary for dependency analysis. The much smaller result is kept in a cache.

Without **open** this would be very easy: All the dependency analyzer needs to know is the set of names that occur free in a source and the set of names defined and exported by the source. In the absence of **open**[6] it is straightforward to calculate these sets for each given source.

With **open** this becomes more involved, but still, the size of typical dependency files is only 1–4% of the corresponding SML source code size.

[6]and the absence of datatype replication as introduced by the revised definition of SML [MTH90, MTHM97]

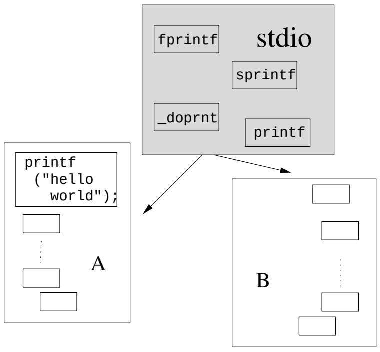

Figure 13: **Controlling sharing using groups.** With a compilation manager like CM we can structure the project shown in figure 2 into a collection of groups. The export interface on group `stdio` will make sure that implementation details such as the local function `_doprnt` are not exposed to its clients A and B. At the same time, since `stdio` is specific in not listing either A or B in its imports it cannot depend on details of one of them that would prevent it from being compatible with the other.

5 Conclusions

CM is now an essential part of SML/NJ. It has been embraced by the user community, because with its group model it enables modular large-scale programming, which normally is difficult in programming environments where one single global namespace is used for all communication between separately compiled sources.

Groups control the effects of external definitions. Any remaining interferences are local and can be solved by modifying only the groups where they occur.

Groups and libraries can be arranged into arbitrarily deep directed acyclic graphs. The resulting hierarchical modularization of the program can be used to great advantage when managing complex programming tasks. For example, `stdio` in figure 2 would be implemented as a group. Other groups from projects A and B would then list it as one of their imports (see figure 13). The `stdio` group specifies its own exports and CM will then guarantee that none of its implementation details are exposed. And since `stdio` does not mention either A or B in *its* imports we are assured that it cannot depend on those.

CM performs well in practice: Analysis time for a typical program is rarely more than 1% of the time required for a full compilation; 0.5% is a typical figure. If only 1% of the program needs to be recompiled, then dependency analysis will account for about 10–20% of the total processing time.

The use of a compilation management tool like CM is considerably simplified by automatic dependency analysis. Unfortunately, some features of the SML language pose as a problem for tractability. We have shown two simple restrictions that fix such problems, and which in our experience are entirely reasonable in practice.

References

[Ada80] Military standard: Ada programming language. Technical Report MIL-STD-1815, Department of Defense, Naval Publications and Forms Center, Philadelphia, PA, 1980.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[AM94] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, volume 29, pages 13–23. ACM Press, June 1994.

[ATW94] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM TOSEM*, 3(1):3–28, January 1994.

[Blu97] Matthias Blume. *Hierarchical Modularity and Intermodular Optimization*. PhD thesis, Princeton University, expected 1997.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 266–277, 1997.

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research Report 52, Systems Research Center, Digital Equipment Corp., Palo Alto, CA, November 1989.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[HLPR94] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. A Compilation Manager for Standard ML of New Jersey. In *1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 136–147, 1994.

[JW78] Kathleen Jensen and Niklaus Wirth. *Pascal: User Manual and Report, Second Ed.* New York, 1978.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, volume 26, pages 293–305. ACM Press, 1991.

[Ten81] R. D. Tennent. *Principles of Programming Languages*. Englewood Cliffs, NJ, 1981.

[Wir82] Niklaus Wirth. *Programming in MODULA-2*. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, second edition, 1982.

[Wir88] Niklaus Wirth. From Modula to Oberon. *Software Practice and Experience*, 18(7), jul 1988.