

# A Generic Framework for Context-Sensitive Analysis of Modular Programs

Germán Puebla<sup>1</sup>, Jesús Correas<sup>1</sup>, Manuel V. Hermenegildo<sup>1,2</sup>, Francisco Bueno<sup>1</sup>, María García de la Banda<sup>3</sup>, Kim Marriott<sup>3</sup>, and Peter J. Stuckey<sup>4</sup>

<sup>1</sup> Department of Computer Science  
Technical University of Madrid (UPM)

<sup>2</sup> Depts. of Computer Science and Electrical and Computer Engineering  
University of New Mexico (UNM)

<sup>3</sup> School of Computer Science and Software Engineering  
Monash University

<sup>4</sup> Department of Computer Science and Software Engineering  
University of Melbourne

{german,jcorreas,herme,bueno}@fi.upm.es  
{mbanda,marrriott}@mail.csse.monash.edu.au  
pjs@cs.mu.oz.au

**Abstract.** Context-sensitive analysis provides information which is potentially more accurate than that provided by context-free analysis. Such information can then be applied in order to validate/debug the program and/or to specialize the program obtaining important improvements. Unfortunately, context-sensitive analysis of modular programs poses important theoretical and practical problems. One solution, used in several proposals, is to resort to context-free analysis. Other proposals do address context-sensitive analysis, but are only applicable when the description domain used satisfies rather restrictive properties. In this paper, we argue that a general framework for context-sensitive analysis of modular programs, i.e., one that allows using all the domains which have proved useful in practice in the non-modular setting, is indeed feasible and very useful. Driven by our experience in the design and implementation of analysis and specialization techniques in the context of CiaoPP, the Ciao system preprocessor, in this paper we discuss a number of design goals for context-sensitive analysis of modular programs as well as the problems which arise in trying to meet these goals. We also provide a high-level description of a framework for analysis of modular programs which does substantially meet these objectives. This framework is generic in that it can be instantiated in different ways in order to adapt to different contexts. Finally, the behavior of the different instantiations w.r.t. the design goals that motivate our work is also discussed.

## 1 Introduction

Analysis of logic programs has received considerable theoretical and practical attention. A number of successful compile-time techniques have been proposed and implemented which allow obtaining useful information on the program and

using such information to debug, validate, and specialize the program, obtaining important improvements in correctness and efficiency. Unfortunately, most of the existing techniques are still only used in prototypes and, though numerous experiments demonstrate their effectiveness, they have not made their way into existing real-life systems. Perhaps one of the reasons for this is that most of these techniques were originally designed to be applied to a complete, monolithic program, while programs in practice invariably have a more complex structure combining a number of user modules with system libraries. Clearly, organizing program code in this modular way has many practical advantages for both program development and maintenance. On the other hand, performing global techniques such as program analysis on modular programs differs from doing so in a monolithic setting in several interesting ways and poses non-trivial problems which must be solved.

In this work we concentrate on *strict* module systems in which procedures external to a module are *visible* to it only if they are part of its *interface*. The interface of a module usually contains the names of the *exported* procedures and the names of the procedures *imported* from other modules. The module can only import procedures which are among the ones exported by the other modules. Procedures which are not exported are not visible outside the module.

Driven by our experience in the design and implementation of context-sensitive analysis and specialization techniques in the CiaoPP system [20, 9], in this paper we present a high level description of a framework for analysis of modular programs. This framework is generic in that it can be instantiated in different ways in order to adapt to different contexts. The correctness, accuracy, and efficiency of the different instantiations is discussed and compared.

The analysis of modular programs has been addressed in a number of previous works. However, most of them have focused on specific analyses with particular properties and using more or less ad-hoc techniques. In [6] a framework is proposed for performing compositional analysis of logic programs in a modular fashion, using the concept of an *open program*, introduced in [2]. An open program is a program in which part of the code is not available to the analyzer. Nevertheless, this interesting framework is valid only for a particular set of abstract domains of analysis—those which are *compositional*.

Another interesting framework for compositional analysis for logic programs is presented in [23], in this case, for *binding-time analysis*. Although the most natural way to describe abstract interpretation-based binding-time analyses is arguably to use a top-down, goal-dependent framework, in this work a goal-independent analysis framework is used in order to simplify the handling of the issues stemming from modularity. The choice is based on the fact that context-sensitivity brings important problems to a top-down analysis framework. Both this paper and [6] stress compositionality as a very attractive property, since it greatly facilitates modular analysis. However, there are many useful abstract domains which do not meet this property, and thus these approaches are not of general applicability.

In [15] a control-flow analysis-based technique is proposed for call graph construction in the context of object oriented languages. Although there has been other work in this area, the novelty of this approach w.r.t. previous proposals is that it is context-sensitive. Also, [1] shows a way to perform modular class analysis by translating the object oriented program into *open* DATALOG programs, in the sense of [2]. These two contributions are tailored to specific analysis domains with particular properties, so an important part of their work is not generally applicable nor reusable in a general framework.

In [21] a two-phase analysis is proposed for incomplete imperative programs, starting with a fast, imprecise global analysis and then continuing with a (possibly context sensitive) analysis for each module in the program. This approach is not abstract interpretation-based. It is interesting to see that it appears to follow from the theory of abstract interpretation that if in such a two-pass approach the first pass “overshoots” the fixed-point, the maximum precision may not be recovered in the second pass.

In [22] a method for performing separate control-flow analysis by means of abstract interpretation is proposed. This paper does not deal with the inter-modular approach studied in the present work, although it does have points in common with our module-aware analysis framework (Section 5). However, in this work the initial information needed by the abstract interpretation-based analyzer is provided by other analysis techniques (types and effects techniques), instead of taking advantage of the actual results from the analysis of the rest of the modules in the program.

A preliminary study of the extension of analysis and specialization to the case of modular programs was presented in [19]. A full practical proposal for modular program analysis was presented in [4], which also presented some preliminary data from its implementation in the context of the Ciao system. Also, an implementation of [4] in the context of the HAL system [8] has been reported in [14].

The rest of the paper proceeds as follows: Section 2 presents a review of program analysis based on abstract interpretation and of the non-modular framework that we use as a starting point. Section 3 then presents some additional notation related to modular programs and a first, simple approach to extending the framework to handling such modular programs: the “flattening” approach. This approach is used as baseline for comparison throughout the rest of the paper. Section 4 then identifies a number of characteristics that are desirable of a modular analysis system and which the simple approach does not meet in general. Achieving (at least a subset of) these characteristics justifies the more involved approach presented in the rest of the paper. To this end, Section 5 first discusses the modifications made to the analysis framework for non-modular programs in order to be able to handle one module at a time. Section 6 then presents the actual full framework for analysis of modular programs. The framework proposed is parametric on the *scheduling policies*. The following sections discuss two scheduling policies which are fundamentally different: *manual scheduling* (Section 7), which corresponds to a scenario where one or more users decide when

and what modules to analyze individually (but in a context-sensitive way), such as in distributed program development, and *automatic scheduling* (Section 8), where a full scheduling policy automatically determines in which order the modules will be analyzed and continues until the process is completed (a fixed-point is reached). Section 9 addresses some practical implementation issues, including persistence and handling of libraries. Finally, Section 10 compares the behavior of the different instantiations of the generic framework proposed together with that of the flattening approach w.r.t. the desirable design features discussed in Section 4, and presents some conclusions.

## 2 A Non-Modular Context-Sensitive Analysis Framework

The aim of context-sensitive program analysis is, for a particular description domain, to take a program and a set of initial call patterns and to annotate the program with information about the current environment at each program point whenever that point is reached when executing calls described by the initial call patterns.

### 2.1 Program Analysis by Abstract Interpretation

Abstract interpretation [7] is a technique for static program analysis in which execution of the program is simulated on a description (or abstract) domain ( $D_\alpha$ ) which is simpler than the actual (or concrete) domain ( $D$ ). Values in the description domain and sets of values in the actual domain are related via a pair of monotonic mappings  $\langle \alpha, \gamma \rangle$ : *abstraction*  $\alpha : 2^D \rightarrow D_\alpha$  and *concretization*  $\gamma : D_\alpha \rightarrow 2^D$  which form a Galois connection, i.e.

$$\forall x \in 2^D : \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall \lambda \in D_\alpha : \alpha(\gamma(\lambda)) = \lambda.$$

The set of all possible descriptions represents a description domain  $D_\alpha$  which is usually a complete lattice or cpo for which all ascending chains are finite. Note that in general  $\sqsubseteq$  is induced by  $\subseteq$  and  $\alpha$  (in such a way that  $\forall \lambda, \lambda' \in D_\alpha : \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$ ). Similarly, the operations of *least upper bound* ( $\sqcup$ ) and *greatest lower bound* ( $\sqcap$ ) mimic those of  $2^D$  in some precise sense. A description  $\lambda \in D_\alpha$  *approximates* a set of concrete values  $x \in 2^D$  if  $\alpha(x) \sqsubseteq \lambda$ . Correctness of abstract interpretation guarantees that the descriptions computed approximate all of the actual values which occur during execution of the program.

Different description domains may be used which capture different properties with different accuracy and cost. Also, for a given description domain, program, and set of initial call patterns there may be many different analysis graphs. However, for a given set of initial call patterns, a program and abstract operations on the descriptions, there is a unique *least analysis graph* which gives the most precise information possible.

## 2.2 The Generic Non-Modular Analysis Framework

We will now briefly describe the main ingredients of a generic context-sensitive analysis framework which computes the least analysis graph. This framework generalizes the particular analysis algorithms used in systems such as PLAI [12, 13], GAIA [5], and the CLP( $\mathcal{R}$ ) analyzer [11], and we believe captures the essence of most context-sensitive, non-modular analysis systems. More details on this generic framework can be found in [10, 17].

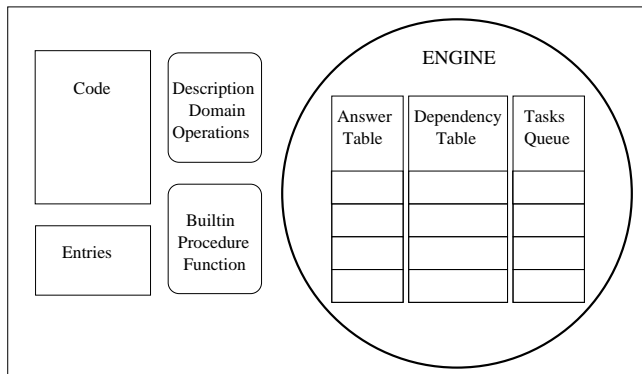
We first introduce some notation.  $CD$  and  $AD$  stand for descriptions in the abstract domain. The expression  $P : CD$  denotes a *call pattern*. This consists of a predicate call together with a call description for that predicate call. Similarly,  $P : AD$  denotes an answer pattern, though it will be referred to as  $AD$  when it is associated to a call pattern  $P : CD$  for the same predicate call.

The least analysis graph for the program is implicitly represented in the algorithm by means of two data structures, the *answer table* and the *dependency table*. Given the information in these data structures it is straightforward to construct the graph and the associated program point annotations. The answer table contains entries of the form  $P : CD \mapsto AD$ . It is interpreted as: the answer pattern for calls of the form  $CD$  to  $P$  is  $AD$ . A dependency is of the form  $P : CD_0 \Rightarrow B_{key} : CD_1$ . This is interpreted as follows: if the procedure  $P$  is called with description  $CD_0$  then this causes the procedure  $B$  to be called with description  $CD_1$ . The subindex *key* can be used in order to uniquely identify the program point within  $P$  where  $B$  is called with calling pattern  $CD_1$ . Dependency arcs represent the arcs in the program analysis graph from procedure calls to the corresponding call pattern.

Intuitively, different analysis algorithms correspond to different graph traversal strategies which place entries in the answer table and dependency table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixed-point algorithms, we use a priority queue. The queue contains the events to process. Different priority strategies correspond to different analysis algorithms. Thus, the third, and final, structure used in our generic framework is a *tasks queue*.

When an event being added to the tasks queue is already in the queue, a single event with the maximum of the priorities is kept in the queue. Also, only one arc of the form  $P : CD \Rightarrow B_{key} : CD'$  for each tuple  $(P, CD, B_{key})$  exists in the dependency table: the last one added. The same holds for entries  $P : CD \mapsto AD$  for each tuple  $(P, CD)$  in the answer table.

Figure 1 shows the architecture of the framework. The *Code* corresponds to the (source) code of the program to be analyzed. By *Entries* we denote the initial starting points for analysis. The box *Description Domain Operations* represents the definition of operations which are domain dependent. The circle represents the *Analysis Engine*, which has the three data-structures mentioned above, i.e., the answer table, the dependency table, and the tasks queue. Initially, for each analysis these three structures are empty and the analysis engine takes care of processing the events on the priority queue by repeatedly removing the highest priority event and calling the appropriate event-handling function. This in



**Fig. 1.** Non-Modular Analysis Framework

turn consults and modifies the contents of the answer and dependency tables. When the tasks queue becomes empty then the analysis engine has reached a fixed-point. This implies that the least analysis graph has been found. We will use  $Analysis_{D_\alpha}(Q, E) = (AT, DT)$  to denote that the analysis of program  $Q$  for initial descriptions  $E$  in domain  $D_\alpha$  produces the answer table  $AT$  with dependency table  $DT$ .

### 2.3 Predefined Procedures

In order to simplify their presentation, formalizations of program analysis often do not consider *predefined* procedures. However, in practice, program analysis implementations allow the use of predefined (language built-in and/or library) procedures<sup>5</sup> in the programs to be analyzed. These *external* procedures whose code is not available in the program being analyzed are often handled in an *ad-hoc* way. Thus, in fairness, non-modular program analyses are more accurately represented by adding to the framework a *builtin procedure function* which essentially hardwires the answer table for these external procedures. This function is represented in Figure 1 by the box *builtin procedure function*. We will use  $\mathcal{CP}$  and  $\mathcal{AP}$  to denote, respectively, the set of all call patterns and the set of all answer patterns. The builtin procedure function can be formalized as a function  $BF : \mathcal{CP} \rightarrow \mathcal{AP}$ . For all call pattern  $P : CD$  where  $P$  is a builtin procedure  $BF(P : CD)$  returns a description  $AD$  which is assumed to be correct in the sense that it is a safe approximation, i.e. an over-approximation of the actual answer pattern for  $P : CD$ .

It is important to note that the data structures which are outside the analysis engine, *code*, *entries*, *description domain operations*, and *builtin procedure function* are read-only. However, though the code and entries are supposed to

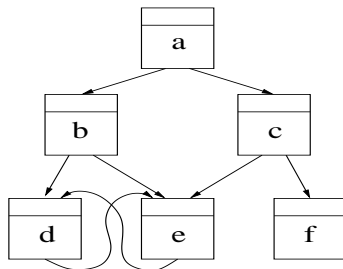
<sup>5</sup> In our modular design, a library can be treated simply as (yet another) module in the program. However, special practical considerations for them will be discussed in Section 9.3.

change for the analysis of each particular program, the *builtin procedure function* can be considered to be fixed, for each description domain  $D_\alpha$ , in that it does not vary from the analysis of one program to another. Indeed, it can be considered to be part of the analyzer. Thus, the builtin procedure function is not explicitly represented as an input to the analysis algorithm.

### 3 The Flattening Approach to Modular Processing

We start by introducing some notation. We will use  $m$  and  $m'$  to denote *modules*. Given a module  $m$ , by  $imports(m)$  we denote the set of modules which  $m$  imports. Figure 2 presents a modular program. Modules are represented as boxes and there is an arrow from  $m$  to  $m'$  iff  $m$  imports  $m'$ . In our example,  $imports(a) = \{b, c\}$ . By  $depends(m)$  we refer to the set generated by the transitive closure of  $imports$ , i.e.  $depends(m)$  is the least set such that  $imports(m) \subseteq depends(m)$  and  $m' \in depends(m)$  implies that  $imports(m') \subseteq depends(m)$ . In our example,  $depends(a) = \{b, c, d, e, f\}$ . Note that there may be circular dependencies among modules. In our example,  $e \in depends(d)$  and  $d \in depends(e)$ . A module  $m$  is a *leaf* if  $depends(m) = \emptyset$ . In our example, the only leaf module is  $f$ . By  $callers(m)$  we denote the set of modules which import  $m$ . In the example,  $callers(e) = \{b, c, d\}$ . Also, we define  $related(m) = callers(m) \cup imports(m)$ . In our example,  $related(b) = \{a, d, e\}$ .

The *program unit* of a given module  $m$  is the finite set of modules containing  $m$  and the modules on which  $m$  depends:  $program\_unit(m) = \{m\} \cup depends(m)$ .  $m$  is called the *top-level* module of its program unit. In our example,  $program\_unit(a) = \{a, b, c, d, e, f\}$  and  $program\_unit(c) = \{c, d, e, f\}$ . A program unit  $U$  is self-contained in the sense that  $\forall m \in U : m' \in imported(m) \rightarrow m' \in U$ .



**Fig. 2.** An Example of Module Dependencies

Several *compilation tasks* such as program analysis and specialization are traditionally considered *global*, as opposed to *local*. Usually, local tasks process one procedure at a time and all the information required for performing the task can be obtained by inspecting that procedure. In contrast, in global tasks the results of processing a part of the program (say, a procedure) may be needed

in order to process other parts of the program. Thus, global processing often requires iterating on the whole program until a fixed-point is reached.

In a modular setting, it may well be the case that part of the information needed to perform the task on (a procedure in) module  $m$  has to be computed in modules other than  $m$ . We will refer to the information originated in modules different from  $m$  as *inter-modular* information in contrast to the information originated in  $m$  itself, which we will call *intra-modular*.

*Example 1.* In context-sensitive program analysis there is an information flow of both call and success patterns to and from procedures in different modules. Thus, program analysis requires inter-modular information. For example, the module  $c$  receives call patterns from module  $a$  since  $callers(c) = \{a\}$ , and it has to propagate the corresponding success patterns to  $a$ . In turn,  $c$  provides  $\{e, f\} = imports(c)$  with call patterns and receives success patterns from them.

### 3.1 Flattening a Program Unit vs. Modular Processing

Applying a framework for non-modular programs to a module  $m$  has the difficulty that  $m$  may not be self-contained. However, there should be no problem in applying the framework if  $m$  is a leaf module. Furthermore, given a global process such as program analysis, at least in principle, it is not obvious that it makes much sense to apply the process to a module  $m$  alone. In principle, it makes more sense to apply it to program units since they are conceptually self-contained. Thus, given a module  $m$  one natural approach seems to be to apply the tool (simultaneously) to all the modules in  $U = program\_unit(m)$ .

Given a program unit  $U$  it is always possible to build a single module  $m_{flat}$  which is equivalent to  $U$  and which is a leaf. The process of constructing such a module  $m_{flat}$  usually only amounts to renaming apart identifiers in the different modules in  $U$  so as to avoid name clashes. We will use  $flatten(U) = m_{flat}$  to denote that the module  $m_{flat}$  is the result of renaming apart the code in each module in  $U$  and concatenating its code into a monolithic module  $m_{flat}$ . This points to a simple solution to the problem of processing modular programs (at least for the case in which all the code is available): to transform  $program\_unit(m)$  into the equivalent monolithic program  $m_{flat}$ . It is then straightforward to apply any tool for non-modular programs to the leaf module  $m_{flat}$ . Figure 3 represents the case in which the non-modular analysis framework is used on the flattened program.

Given the existence of an implementation for non-modular analysis, this approach is often simple to apply. Also, this flattening approach has theoretical interest. It can be used, for example, in order to compare the efficiency of different approaches to modular handling of programs w.r.t. the flattening approach. However, as a practical way in which to actually perform analysis of program units this approach has important drawbacks. This issue will be discussed in more detail in Section 10.

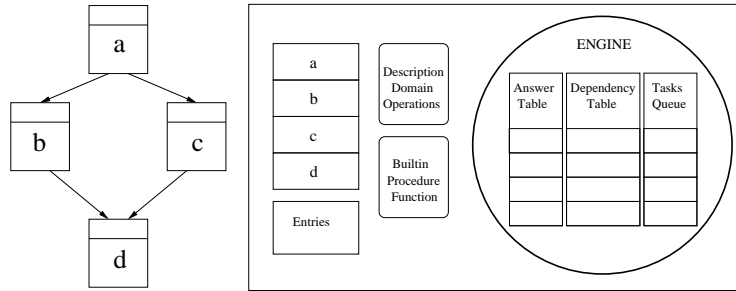


Fig. 3. Using non-modular analysis on a flattened program

## 4 Design Goals for Analysis of Modular Programs

Before presenting our proposals for analysis of modular programs, we will discuss the main features which should be taken into account when designing and/or implementing a tool for context-sensitive analysis of modular programs. As often happens in practice, some of the features presented are conflicting with others and this might make it impossible to find a framework which behaves optimally w.r.t. all of them.

*Module-Awareness* We consider a framework *module-aware* when it has been designed with modules in mind. Thus, it is applicable to a module  $m$  by using the code of  $m$  and some “interface” information for the modules in  $imports(m)$ . Such interface information will in general consist of a summary of previous analysis results for such modules, if such results are available, or a safe approximation if they are not.

Though transforming a non-modular framework into a module-aware one may seem trivial, it requires identifying precisely which is the required information on the result of applying the tool in each of the modules in  $imports(m)$  which should be stored in order to apply the tool to  $m$ . This corresponds in general to the inter-modular information. It is also desirable that the amount of such information be minimal.

*Example 2.* The framework for non-modular analysis in Section 2 is indeed non-modular since it requires the code of all procedures (except possibly for some predefined ones) to be available to the analyzer. It will produce wrong results when applied to non-leaf modules since a missing procedure can only be deemed as an error, unless the framework is aware that such a procedure can be imported.

*Correctness* The results of applying the tool to a module  $m$  should produce results which are *correct*. The notion of correctness itself can in general be lifted from the non-modular case to the modular case without great difficulties. A more complex issue is how to extend a framework to the modular case in such a way that correctness is preserved.

*Accuracy* Similarly, the analysis results for a module  $m$  should be as accurate as possible. The notion of accuracy can be defined by comparing the analysis results with those which would be obtained using the flattening approach presented in Section 3.1 above, since the latter always computes the most accurate information possible, which corresponds to the least analysis graph.

*Termination* A framework for analysis of modular programs should guarantee termination (at least) in all cases in which the flattening approach terminates (which, typically, is for every program). Such termination is guaranteed by choosing description domains with some specific characteristics such as having finite height, finite ascending chains, etc., and/or incorporating a *widening operator*.

*Efficiency in Time* The time required to apply the tool should be reasonable. We will understand “reasonable” as not over an acceptable threshold on the time taken using the flattening approach.

*Efficiency in Memory* In general, one of the main expected advantages of the modular approach is that the total amount of memory required to handle each module separately should be smaller than that needed in the flattening approach.

*No Need for Analyzing All Call Patterns* Under certain circumstances, applying a tool on a module  $m$  may require processing only a subset of the call patterns rather than all call patterns for  $m$ . In order to achieve this, the model must keep track of fine-grained dependencies. This will allow marking exactly those call patterns which need processing. Other call patterns not marked do not need to be processed.

*Support for the Co-Existence of Multiple Program Units/Applications* In a modular setting it is often the case that a particular module is used in several applications. Support for software reuse is thus a desirable feature. However, this poses additional and interesting challenges to the tools, some of which will be discussed in Section 9.

*Support for Source Changes* What happens if the source of a module changes during processing? Some tools will not allow this at all and if it happens all the processing has to start again from scratch. This has the disadvantage that the tool is then not incremental since a (possibly minor) change in a module invalidates the information for all the program unit. Other tools may delete the information which may depend on the changed code, but still keep the information which does not depend on it.

*Persistence* This feature indicates that the inter-modular information can be stored in a persistent medium, such as a file stored on disk or a database, and allow later recovery of such information.

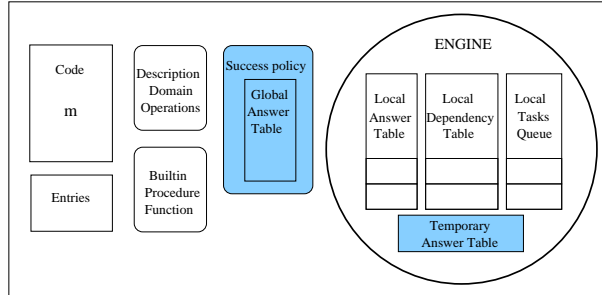


Fig. 4. Module-aware analysis framework

## 5 Analysis of Modular Programs: The Local Level

As a first step towards introducing our analysis framework for modular programs, which will be presented in Section 6 below, in this section we discuss the main ingredients which have to be added to an analysis framework for non-modular programs in order to be able to handle one module at a time.

Analyzing a module separately presents the difficulty that, from the point of view of analysis, the code to be analyzed is *incomplete* in the sense that the code for procedures imported from other modules is not available to analysis. More precisely, during analysis of a module  $m$  there may be calls  $P : CD$  such that the procedure  $P$  is not defined in  $m$  but instead it is imported from another module  $m' \in \text{imports}(m)$ . We refer to determining the value of  $AD$  to be used for  $P : CD \mapsto AD$  as the *imported success problem*. In addition, in order to obtain analysis information for  $m'$  which is as accurate as possible we need to somehow propagate the call  $P : CD$  to  $m'$  so that the next time  $m'$  is analyzed such a call pattern is taken into account. We refer to this as the *imported calls problem*. Note that in this case analysis has to be module-aware in order to determine whether a given procedure is either local or imported (or predefined).

Figure 4 shows the architecture of an analysis framework which is module-aware. This framework is an extension of the non-modular framework in Figure 1. One minor change is that the read/write data structures internal to the analysis engine have been renamed with the prefix “local”. So now we have the *local answer table*, the *local dependency table*, and the *local task queue*. Also, the box which represents the code now contains  $m$  indicating that it contains the single module  $m$ .

The shaded boxes in Figure 4 indicate the main differences w.r.t. the non-modular framework. One is that in the module-aware framework there is an additional read-only<sup>6</sup> data structure, the *global answer table*, or *GAT* for short. Its contents are identical in format to those in the answer table of the non-modular framework. There are however some differences: (1) the *GAT* contains analysis

<sup>6</sup> In fact, this data structure is read/write at the global level discussed in Section 6 below, but it is read-only as regards our engine for analysis of one module.

results which were obtained previously to the current analysis step. (2) The *GAT* contains entries which correspond to predicates defined in  $\text{imports}(m)$ , whereas all entries in the local answer table (or *LAT* for short) are for predicates defined in  $m$  itself. (3) Only information of exported predicates is available in *GAT*. The *LAT* contains information for all predicates in  $m$  regardless of whether they are exported or not.

## 5.1 Solving the Imported Success Problem

The second important difference is that the module-aware framework requires the use of a *success policy*, or *SP* for short, which is represented in Figure 4 with a shaded box surrounding the *GAT*. The *SP* can be seen as an intermediary between the *GAT* and the analysis engine. The behavior of the analysis engine for predicates defined in  $m$  remains exactly as before. *SP* is needed because though the information in the *GAT* will be used in order to obtain answer patterns for imported predicates, given a call pattern  $P : CD$  it will often be the case that an entry of exactly the form  $P : CD \mapsto AD$  does not exist in *GAT*. In such case, the information already present in *GAT* may be of value in order to obtain a (temporary) answer pattern  $AD$ . Note that the *GAT* together with *SP* will allow solving the “imported success problem”.

In contrast, in many formalizations of non-modular analysis there is no explicit success policy. This is because if the call pattern  $P : CD$  has not been analyzed yet, the analysis algorithm forces its computation. Thus, the results of analysis do not depend on any particular success policy: when analysis reaches a fixed-point there is always an entry of the form  $P : CD \mapsto AD$  for any call pattern  $P : CD$  which appears in the analysis graph. Unfortunately, in a modular setting it is not directly possible to force the analysis of predicates defined in other modules. Those modules may have already been analyzed or they may be analyzed in the future. We will simply do what we can given the information available in *GAT*.

We will use  $\mathcal{GAT}$  to denote the set of all global answer tables. The success policy can be formalized as a function  $SP : \mathcal{CP} \times \mathcal{GAT} \rightarrow \mathcal{AP}$ . Several success policies can be defined which provide over- or under-approximations of the exact answer pattern  $AD^\#$  with different degree of accuracy. Note that this exact value  $AD^\#$  is the one which the flattening approach would compute. In this work we consider two kinds of success policies, those which are guaranteed to always provide over-approximations, i.e.  $AD^\# \sqsubseteq SP(P : CD, AT)$ , and those which provide under-approximations, i.e.,  $SP(P : CD, AT) \sqsubseteq AD^\#$ . We will use the superscript  $+$  (resp  $-$ ) to indicate that a success policy over-approximates (resp. under-approximates). As will be discussed later in the paper, both over- and under-approximations are useful in different contexts and for different purposes. Since it is always required to know whether a success policy over- or under-approximates we will mark all success policies in either way.

*Example 3.* A very precise over-approximating success policy is the function  $SP_{All}^+$  defined below, already proposed in [19]:

$$SP_{All}^+(P : CD, GAT) = \text{topmost}(CD) \sqcap_{AD' \in \text{app}} AD' \text{ where}$$

$$\text{app} = \{AD' \mid (P : CD' \mapsto AD') \in GAT \text{ and } CD \sqsubseteq CD'\}$$

The function *topmost* obtains the topmost answer pattern for a call pattern. The notion of *topmost description* was already introduced in [3]. Informally, a topmost description keeps those properties which are *downwards closed* whereas it loses those ones which are not. Note that taking  $\top$  as answer pattern is a correct over-approximation, but often less accurate than using topmost substitutions. For example, if a variable is known to be ground in the call pattern, it will continue being ground in the answer pattern and taking  $\top$  as the answer pattern would lose this information. However, the fact that a variable is free on call does not guarantee that it will keep on being free on success.

We refer to this success policy as  $SP_{All}^+$  because it uses *all* entries in  $GAT$  which are *applicable* to the call pattern in the sense that the call pattern already computed is more general than the call being analyzed.

*Example 4.* The counter-part of  $SP_{All}^+$  is the function  $SP_{All}^-$  which is defined as:

$$SP_{All}^-(P : CD, GAT) = \sqcup_{AD' \in \text{app}} AD' \text{ where}$$

$$\text{app} = \{AD' \mid (P : CD' \mapsto AD') \in GAT \text{ and } CD' \sqsubseteq CD\}$$

Note the change in the direction of the applicability relation (the call pattern in the  $GAT$  has to be more particular than the one being analyzed) and the use of the lub operator instead of the glb. Also, note that taking, for example,  $\perp$  as an under-approximation is correct but  $SP_{All}^-$  is more precise.

## 5.2 Solving the Imported Calls Problem

The third important difference w.r.t. the non-modular framework is the use of the *temporary answer table* (or  $TAT$  for short) and which is represented as a shaded box within the analysis engine of Figure 4. This answer table will be used to store call patterns for imported predicates which are not yet present in  $GAT$  and whose answer pattern has been obtained (approximated) using the success policy on the entries currently stored in  $GAT$ . The  $TAT$  is used as a cache for imported call patterns and their corresponding answer patterns, thus avoiding having to repeatedly apply the success policy on the  $GAT$  for equivalent call patterns, which is an expensive operation. Also, after analysis of the current module is finished, the existence of the  $TAT$  simplifies the way in which the global data structures need to be updated. This will be discussed in more detail in Section 6 below.

We use  $MAnalysis_{D_\alpha}(m, E_m, SP, GAT) = (LAT_m, LDT_m, TAT_M)$  to denote that the module-aware analysis framework returns  $(LAT_m, LDT_m, TAT_M)$  when applied to module  $m$  for initial call patterns  $E_m$  with  $SP$  and  $GAT$ .

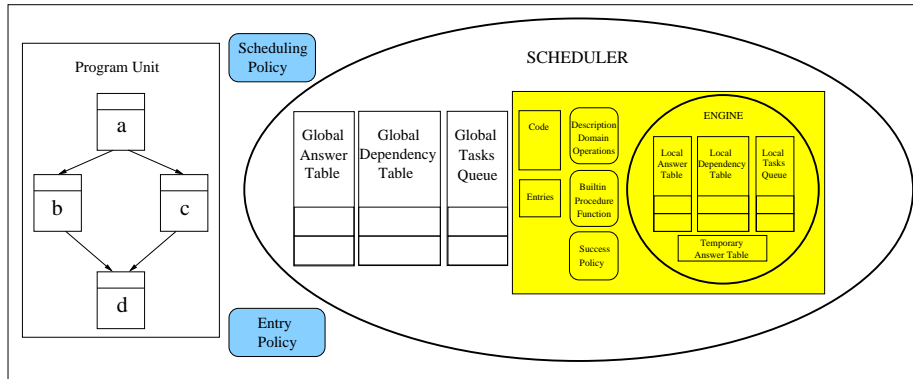


Fig. 5. A two-level framework for analysis of modular programs

## 6 Analysis of Modular Programs: The Global Level

After discussing the *local-level* issues which appear when analyzing a module, in this section we present a complete framework for the analysis of modular programs. Since analysis is a global task, an analysis framework should not only deal with local-level information, but also with global-level information. A graphical representation of our framework is depicted in Figure 5. The main idea is that we have to add a higher-level component to the framework which takes care of the *inter-modular* information, as opposed to the *intra-modular* information which is handled by the local-level subsystem described in the previous section.

As a result, analysis of modular programs is best seen as a two-level process. Note that the inner, lightly shaded, rectangle corresponds exactly to Figure 4 as it is a module-aware analysis system. It is interesting to see how the data structures in the global and local levels are indeed very similar. The similarities and differences between the *GAT* and *LAT* have been discussed already in Section 5 above. Regarding the global and local dependency tables (*GDT* and *LDT* respectively), they are used in order to be able to propagate as precisely as possible which parts of the analysis graph have to be recomputed. The *GDT* is used in order to add events to the global task queue (*GTQ*) whereas the *LDT* is used to add events (*arcs*) to be (re-)analyzed to the local task queue (*LTQ*). We can define the events to be processed at the global level using different levels of granularity. As usual, the finer-grained these events are, the more detailed and thus more effective the handling of the events can be. One obvious possibility is to use modules as events. This means that all call patterns which correspond to a module are handled simultaneously whenever the module is selected at the global level. A more refined possibility is to keep events at the call pattern level. This, together with sufficiently detailed information in the *GDT* will allow incrementality at the call pattern level rather than module level.

## 6.1 Parameters of the Framework

The framework has three parameters. The *program unit* corresponds to the program unit to be analyzed. Note that the code may not be physically stored in the tool's memory since it is already on external storage. However, the framework may maintain some information on the program unit, such as dependencies among modules, *strongly connected components*, and any other information which may be useful in order to guide analysis. In the figure the *program unit* is represented, as an example, containing a program unit composed of four modules. The second parameter is the *entry policy*, which determines the way in which the *GTQ* and *GAT* should be initialized whenever analysis of a program unit is started. Depending on how the success policy is defined, entries for all procedures exported in each of the modules in the program unit may be required in *GAT* and *GTQ* or not.

Finally, the *scheduling policy* determines the order in which the entries in the *GTQ* should be processed. The efficiency with which the fixed-point is reached can differ very much from some scheduling policies to others. Since the framework presented in Figure 5 has just one analysis engine, processing a call pattern in a different module from that currently loaded has a relevant cost associated to it, since this often requires context switching from the current module to a new module. Thus, it is often a good idea to process all or many of the call patterns in *GTQ* which correspond to the module which is being analyzed in order to minimize the number of times the analysis tool has to switch from one module to another. In the rest of the paper we consider that events in *GTQ* are answer patterns which would benefit from (re-)analysis. The role of the scheduling policy is to select a set of patterns from *GTQ* which must necessarily belong to the same module  $m$  to be analyzed. Note that a scheduling policy based on modules can always be obtained by simply processing at each analysis step all events in *GTQ* which correspond to  $m$ .

## 6.2 How the Global Level Works

As already mentioned, analysis of a modular program starts by initializing the global data structures as indicated by the entry policy. At each step, the scheduling policy is used to determine the set  $E_m$  of entries for module  $m$  which are to be processed. They are removed from *GTQ* and copied into the data structure *Entries*. The code of the module  $m$  is also copied to *code*. Then,  $MAnalysis(m, E_m, SP) = (LAT_m, LDT_m, TAT_m)$  is computed. Then, the global data structures are updated, as detailed in Section 6.3 below. As a result of this, new events may be added to *GTQ*. Analysis terminates when there are no more events to process in *GTQ* or when the scheduling strategy does not select any further events.

Each entry in *GTQ* is of one of the following three types: *over-approximation*, *under-approximation*, or *invalid*, according to the reason why they should be re-analyzed. An entry  $P : CP \mapsto AP$  which is an over-approximation is marked  $P : CP \mapsto^+ AP$ . This indicates that the answer pattern  $AP$  is possibly an

over-approximation since it depends on a call pattern whose answer pattern has been determined to be an over-approximation. In other words, the accuracy of  $P : CP \mapsto AP$  may be improved by re-analysis. Similarly, under-approximations are marked  $P : CP \mapsto^- AP$  and they indicate that  $AP$  is probably an under-approximation since it depends on a call pattern whose success pattern has increased. As a result, the call pattern should be re-analyzed to guarantee correctness. Finally invalid entries are marked  $P : CP \mapsto^\perp AP$ . They indicate that the relation between the current answer pattern  $AP$  and one resulting from re-computing it for  $P : CP$  is unpredictable. This often indicates that the source code of the module has changed in a way that the analysis results for some of the exported procedures are just incompatible with previous ones. Handling this kind of events is discussed in more detail in Section 6.4 below.

### 6.3 Updating the Global State

In Section 5 it has been presented how the local level subsystem, given a module  $m$ , can compute the corresponding  $LAT_m$ ,  $LDT_m$ , and  $TAT_m$ . However, once analysis of module  $m$  is done, the analysis results of module  $m$  have to be used in order to update the global state prior to starting analysis of any other module.

We now briefly discuss how this updating is done. For each initial call pattern  $P : CP$  in *Entries* we compare the previous answer pattern  $AP$  with the newly computed one  $AP'$ . If  $AP = AP'$  then this call pattern has not been affected by the latest analysis. However, it is also possible that the answer pattern “evolves” in different analysis iterations. If we use  $SP^+$ , the natural thing is that the new answer pattern is more specific than the previous one, i.e.,  $AP' \sqsubset AP$ . In such case those call patterns which depend on  $P : CP$  can also improve their success pattern. We use the *GDT* to locate all such patterns and we add them to the *GTQ* with the  $+$  mark. Conversely, if we use  $SP^-$ , the natural thing is that  $AP \sqsubset AP'$ . We then add events marked  $-$ .

In a typical situation, and if modules do not change, all events in *GTQ* will be approximations of the same sign. This depends on the success policy used. If the success policy is of kind  $SP^+$  (resp.  $SP^-$ ) then the events which will be added to *GTQ* will also be over-approximations (resp. under-approximations). In turn, when they are processed they will introduce other over-approximations (resp. under-approximations).

The  $TAT_m$  is also used to update the global state. All entries in  $TAT_m$  are added to *GAT* and *GTQ* marked with the same sign as the success policy used. Last, we also have to update the *GDT*. For this, we first erase all entries for any of the call patterns which we have just analyzed, and which are thus stored in  $entries_m$ . Then we add an entry of the form  $P : CP \rightarrow H : CP'$  for each imported procedure  $H$  which is reachable with call pattern  $CP'$  from an initial call pattern  $P : CP$ . Note that this can easily be determined using *LDT*.

## 6.4 Recovering from an Invalid State

If code of a module  $m$  has changed since it was last analyzed, it can be the case that the global information available is invalid. This happens when in the results of re-analysis of  $m$  any of the exported predicates has an answer pattern which is incompatible with the previous results. In this case, all information dependent on the new answer patterns might have become invalid, as discussed in Section 6.2. The question is how to minimize the impact of such a situation.

The simplest solution is to (transitively) erase any information of other modules which depends on the invalidated one. This solution may not be very efficient, as it ignores all results of previous analyses of other modules even if the changes performed in the module are minor, or only affect directly related modules. Another alternative is to launch an automatic recovery process as soon as invalid analysis results are detected (see [4]). This process has to reanalyze the modules directly affected by the invalidated answer pattern(s). If the new answer patterns coincide with the old ones then the changes do not affect this module and the process terminates. Otherwise, it continues transitively with the directly related modules.

## 7 Using a Manual Scheduling Policy

Consider, for example, the relevant case of independent development of different parts of the program, which can then even be performed in parallel by different teams. In this setting, it makes sense that the analyzer performs its job on the current module without analyzing other modules in the program unit, i.e., it allows separate analysis. This will typically allow early detection of compile-time errors in the current module without having to wait for the code of the dependent modules to be fully developed. Moreover, in this setting, it is the user (or users) who decide when and what to analyze. Thus, we refer to this as the *manual* setting. Furthermore, we assume that in this setting analysis for a module  $m$  has to do its best with only the code for  $m$  plus the results of previous analyses (if any) of the modules in  $depends(m)$ . These assumptions have important implications. The setting allows the users of different modules to decide when they should be processed. And thus, any module could be (re-)analyzed at any point. As a result, strong requirements must hold for the whole approach to be correct. In return, the results obtained may not be optimal (in terms of error detection, degree of optimization, etc., depending on the particular tools) w.r.t. those achievable using automatic scheduling.

So the question is, is there any combination of the three parameters of the global analysis framework which allows handling the manual setting? The answer to this question is yes. Our earlier paper [4] essentially describes such an instantiation of the analysis framework. In the terminology of the current paper, the model in [4] corresponds to waiting until the user requests that a module  $m$  in the program unit  $U$  be analyzed. The success policy is over-approximating. This guarantees that in the absence of invalidated entries in the  $GTQ$  all events will be marked  $+$ . This means that the analysis information available is correct,

though perhaps not as accurate as possible. Since the scheduling is manual, no other analyses should be triggered until the user requires so. Finally, the entry policy is simply to include in *GTQ* an event such as  $P : \top \mapsto^+ \top$  per predicate exported by any of the modules in  $U$  to be analyzed (it is called *all* entry policy). The initial events are required to be so general to keep the overall correctness of the analysis while allowing the users to choose the order of the modules to be analyzed.<sup>7</sup> The model in [4] has the very important feature of being guaranteed to always provide correct results without the need of reaching a global fixed-point.

## 8 Using an Automatic Scheduling Policy

In spite of the evident interest of the manual setting, there are situations in which the user is interested in obtaining the most accurate analysis results possible. For this, it may be required to analyze the modules in the program unit several times in order to converge to a distributed global fixed-point. We will refer to this as the *automatic* setting, in which the user decides when to start global analysis of a program unit. From then on it is the global analysis framework by means of its *scheduling policy* who decides when and what to analyze. Note that the manual and automatic settings roughly correspond to scenario 1 and scenario 2 of [19] respectively. Since we admit circular dependencies among modules, the strategy has to be able to deal with such circularities correctly and efficiently without entering infinite loops. The question now is what are the values for the different parameters to our generic framework which should be used in order to obtain satisfactory results? One major difference of the automatic setting w.r.t. the manual setting is that in addition to over-approximations, now also under-approximations can be used. This is because though under-approximations do not guarantee correctness in general, when an inter-modular fixed-point is reached, analysis results are guaranteed to be correct. Below we consider the use of  $SP^+$  and  $SP^-$  separately.

### 8.1 Using Over-Approximating Success Policies

If a success policy  $SP^+$  is used, we are in a situation similar to the one in Section 7 in that independently of how many times each module has been analyzed, if there have not been any code changes, the analysis results are guaranteed to be correct. The main difference is that now the system keeps on automatically requesting further analysis steps until a fixed-point is reached.

Regarding the entry policy, an important observation is that in the automatic mode, much as in the case of intra-modular analysis, inter-modular analysis will eventually compute all call patterns which are needed in order to obtain information which is correct w.r.t. calls, i.e., the set of computed call patterns

---

<sup>7</sup> In the case of the Ciao system it is possible to use *entry* declarations (see for example [16]) in order to improve the set of initial call patterns for analysis.

covers all possible calls which may occur at run-time for the class of initial calls considered, i.e., those for the top-level of the program unit  $U$ . This will allow us to use a different entry policy from that used in the manual mode: rather than introducing events of the form  $P : \top \mapsto^+ \top$  in the  $GTQ$  for exported predicates in all modules in  $U$ , it suffices to introduce them for predicates exported by the top-level of  $U$  (this entry policy is named *top-level* entry policy). This has several important advantages: (1) It avoids analyzing all predicates for the most general call pattern, since this may end up introducing plenty of call patterns which are not used in our particular program unit  $U$ . (2) It will help to have a more guided scheduling policy since there are no requests for processing a module until it is certain that a call pattern should be analyzed. (3) If multiple specialization is being performed based on the set of call patterns for each procedure (possibly preceded by a minimization step for eliminating useless versions [18]), the fact that a call pattern with the most general call pattern exists implies that a non-optimized version of the predicate must always exist. Another way out of this problem is to eliminate useless call patterns once an inter-modular fixed-point has been reached.

Since reaching a global fixed-point can be a costly task, one interesting possibility can be the introduction of a time-out. The user can ask the system to request (re-)analysis as needed towards improving the analysis information. However, if after performing  $n$  analysis steps the time-out is reached before analysis  $n + 1$  is finished, the global state corresponding to state  $n$  is guaranteed to be correct. In this case, the entry policy used has to be to introduce most general call patterns for all exported predicates, either before starting analysis or when a time-out is reached.

## 8.2 Using Under-Approximating Success Policies

Another alternative is to use  $SP^-$ . As a result, the analysis results are not guaranteed to be correct until an inter-modular fixed-point is reached. Thus, it may take a large amount of time to perform this global analysis. On the other hand, once a fixed-point is reached, the accuracy which will be obtained is optimal, since it corresponds to the least analysis graph, which is exactly the same which the flattening approach would have obtained.

Regarding the entry policy, the same discussion as above applies. The only difference being that the  $GTQ$  should be initialized with events of the form  $P : \top \mapsto^- \perp$  since now the framework computes under-approximations. Clearly,  $\perp$  is an under-approximation of any description.

Another important thing to note is that, since the final results of automatic analysis are optimal, they do not depend on the use of a particular success policy  $SP_1^-$  or another  $SP_2^-$ . Of course, the efficiency using  $SP_1^-$  can be very different from that obtained using  $SP_2^-$ .

### 8.3 Hybrid policy

In practice we may wish to use a manual scheduling policy with an over-approximating success policy during program development, and then use an automatic scheduling policy with an under-approximating success policy just before program release, so as to ensure that the analysis is as precise as possible, thus allowing as much optimization as possible in the final version.

Fortunately, in such a situation we can often reuse much of the analysis information obtained using the over-approximating success policy. The reason is that if the analysis with the over-approximating success policy has reached a fixed-point, the answers obtained for module  $m$  are as accurate as those obtained with an under-approximating success policy as long as there are no cyclic dependencies between the modules in  $depends(m)$ . Thus in the common case that no modules are mutually dependent we can simply use the answer tables from the manual scheduling policy and use an automatic scheduling policy with an over-approximating success policy to obtain the fixed-point. Even in the case that some modules are mutually dependent we can use this technique to compute the answers for the modules which do not contain cyclic dependencies or do not depend on modules that contain them (e.g., leaf-modules).

### 8.4 Computation of an Intermodular Fixed-Point

Determining the optimal order in which the different modules in the program unit should be analyzed in order to get to a fixed-point as efficiently as possible is not trivial and it is the topic of ongoing work.

Finding good scheduling strategies for intra-modular analysis is a topic which has received considerable attention and highly optimized algorithms exist which converge to a fixed-point quickly. Unfortunately, it is not possible to directly translate the same heuristics used in the intra-modular case to the inter-modular case. In the inter-modular case we have to take into account the time required to change from analysis of one module to another since this typically means reading a new module from disk. Thus, requests to process call patterns have to be grouped by modules in order to reduce the number of times we change context.

Taking the heuristics in [17, 10] as a starting point we are investigating and experimenting with different scheduling policies which take into account different aspects of the structure of the program unit such as dependencies, strongly connected components, etc. with promising results. It also remains to be explored which of the approaches to success policy results in more efficiently reaching a global fixed-point and whether the heuristics to be applied in either case coincide or are mostly different.

## 9 Some Practical Implementation Issues

In this section we discuss several issues not addressed in the previous sections and which are very important in order to have practical implementations of

context-sensitive analysis systems. These issues are related to the persistence of global information and the analysis of libraries.

### 9.1 Making Global Information Persistent

The two-level framework presented in Section 6 needs to keep information both at the local and global level. One relevant question, due to its practical implications, is where this global information actually resides. One possibility is to have the global analysis tool running continuously as a kind of “compilation server” which stores the global state in its program memory. In a *manual* setting, this global tool would wait for the user(s) to place requests to analyze modules. When a request is received, the corresponding module is analyzed for the appropriate call patterns and using the global information available at the time in the memory of the global analyzer. After analysis terminates, the global information is updated and remembered by the process for subsequent requests. If we are in an *automatic* setting, the global tool itself requests the analysis of different modules until a global fixed-point (or a time-out) is reached.

This approach outlined above is not fully persistent in the sense that if the computer crashes all information about the global state is lost and analysis would have to start from scratch again. In order to implement the more general kind of persistence discussed in Section 4, a way to save and restore the global state of analysis is needed. This requires storing the value of the three global-level data-structures: *GAT*, *GDT*, and *GTQ*. A level of granularity which seems appropriate in this context is clearly the module level. I.e., the global state of analysis is saved and restored between two consecutive steps of (module) analysis, but not during the analysis of a given module, which, from the point of view of the two-level framework, is an atomic operation.

The ability to save and restore the global state of analysis has several advantages:

1. The global tool does not need to be running continuously: it can save its state, stop, restart when needed, and restore the global state. This is specially interesting when using a manual scheduling policy, since two consecutive analysis requests can be separated by large intervals.
2. Even if the automatic scheduling policy is used, any information about the global state which is still valid can be directly used. This means that analysis can be *incremental* in the sense that (global level) analysis information which is known to be valid is reused.

### 9.2 Splitting Global Information

Consider the analysis of module  $b$  in the program unit  $U = \{a, b, c, d, e, f, g, h\}$  depicted in Figure 6. In principle, the global state includes information regarding exported predicates in any of the modules in  $U$ . As a result, if we can save the global state to disk and restore it, this would involve storing and retrieving information about all modules in  $U$ . However, analysis of  $b$  only requires retrieving the information for modules in *related*( $m$ ). The small boxes which appear on

the side of every module represent the portion of the global structures related to each module. To analyze the module  $b$ , the information of the global tables that we need is that of modules  $a$ ,  $d$  and  $e$ , as indicated by the dashed curved line.

This is straightforward to do in practice by splitting the information in the global data structures into several parts, each one associated to a module. This allows easily identifying the pieces of global information which are needed in order to process a given module.

This optimization of the handling of global information has several advantages:

1. The time required to save and restore the information to disk is reduced since the total amount of information transferred is smaller.
2. The use of the data structures during analysis can be more efficient since search space is reduced.
3. The total amount of memory required in order to analyze a module can be significantly reduced: only the local data structures plus a possibly very reduced part of the global data structures are actually required to analyze the module.

One question which we have intentionally left open is where the persistent information should reside. In fact, all the discussion above is independent on how and where the global state is stored, as long as it is persistent. One possibility is to use a database which stores the global state and information is grouped by modules in order to minimize the amount of information which has to be retrieved or updated for each analysis. Another, very common, possibility is to store the global information associated to each module to disk, in the same way as temporary information (such as relocatable code) is stored in many traditional compilers. In fact, the actual implementation of modular analysis in both CiaoPP and HAL [14] systems is based on this idea: a module  $m$  has a `m.reg` file associated to it which contains the part of the global data structures which are associated to  $m$ .

### 9.3 Handling Libraries and Predefined Modules

Many compilers and program development systems include a large number of predefined modules and libraries which can be readily reused by programmers –an obviously interesting feature since it greatly reduces the time required to develop applications. From the point of view of analysis, these predefined modules and libraries differ from user programs in a number of ways:

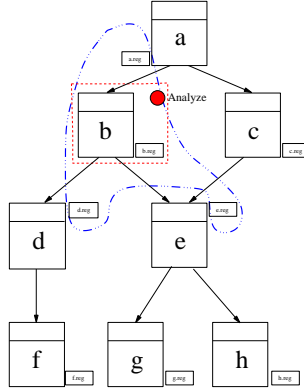
1. They are designed with reusability in mind and thus they can be used by a comparatively large number of user programs.
2. Sometimes the source code for libraries and predefined modules may not be available. One common reason for this is that they are implemented in a lower-level language.

3. The total amount of code available as libraries can be extremely large. Thus, reanalyzing the libraries over and over again for slightly different call patterns can be costly.

Given these characteristics, it makes sense to develop a specialized treatment for libraries. We propose the following scheme. For each library module, the analysis results for a sufficient set of call patterns should be precomputed. This set should cover all possible correct call patterns for the library. In addition, the answer pattern for those call patterns have to be an over-approximation of the actual answers, independently of whether a  $SP^+$  or  $SP^-$  success policy is used for the programs which use such library. In addition, in order to provide more accurate information, more particular call patterns which are expected to occur often in programs which use that library module can also be included. This information is added to the *GAT* of the program units which use the library. Thus, the success policy will be able to use this information directly for obtaining answer patterns. The reason for requiring pre-computed answer patterns for library modules to be over-approximations is that, much in the same way as for predefined procedures, even if an automatic scheduling policy is used, library modules are (in principle) not analyzed for calling patterns other than those which are pre-computed. Note that this is conceptually equivalent to considering the interface information of library modules *read-only*, since any program using them can read this information, but no additional call patterns will be analyzed. As a result, the global level framework will ignore new call patterns to library procedures that might be generated during the analysis of user programs. More precisely, entries of the form  $P : CP \mapsto AP$  in *TAT* such that  $P$  is a library predicate do not need to be added to the *GTQ* since they will not be analyzed. In addition, no entries of the form  $P : CP \rightarrow H : CP'$  need be added to *GDT* if  $H$  is a library predicate, since the answer pattern for library predicates is never modified and thus those dependencies are useless.

Deciding which is the best set of call patterns for which a library module should be analyzed is a non-trivial problem. One possibility can be to extract call patterns from correct programs which use the library and study which are the call patterns most often used. Another possibility is to have the library developer decide which are the call patterns of interest.

In spite of the considerations above, it is sometimes the case that we are interested in treating a library module using the general scheme, i.e., effectively considering the library information writable and allowing the analysis of new call patterns and the storage of the corresponding results. This can be interesting if the source code of a library is available and the set of initial call patterns for which it has been analyzed is not very representative. Note that hopefully this will happen often only when the library is relatively new. Once the code of the library stabilizes and a good set of initial patterns is obtained, it will generally be considered read-only. Allowing reanalysis of a library can also be useful when we are interested in using the analysis results from such call patterns to optimize the code of the library for the particular cases that correspond to those calls.



**Fig. 6.** Using Distributed Scheduling and Local Data Structures

For this case it may be interesting to store the corresponding information locally to the calling module, as opposed to inserting it into the library directories.

In summary, the implementation of the framework needs to treat libraries in a special way and also allow applying the general scheme for some designated library modules.

## 10 Discussion and Conclusions

Table 1 summarizes some characteristics of the different instantiations of the generic framework presented in the paper, in terms of the design features discussed in Section 4. The corresponding entries for the flattening approach of Section 3 –our baseline as usual– are also provided for comparison, listed in the column labeled **Flattening**. The **Manual** column lists the characteristics of the manual scheduling policy described in Section 7. The last two columns correspond to the two instantiations of the automatic scheduling policy, which were presented in Sections 8.1 and 8.2 respectively. **Automatic<sup>+</sup>** (resp. **Automatic<sup>-</sup>**) indicate that an over-approximating (resp. under-approximating) success policy is used.

The first three rows, i.e., **Scheduling policy**, **Success policy**, and **Entry policy** correspond to the values of these parameters in each instantiation.

All instances of the framework for modular analysis are *module-aware*, in contrast to **Flattening**, which is not. Both instances described of the modular framework proposed are incremental, in the sense that only a subset (instead of every module) in the program unit needs to be re-analyzed, and they also both achieve the goal of *not needing to reanalyze all call patterns* every time a module is considered for analysis.

Regarding correctness, both the **Flattening** and **Automatic<sup>-</sup>** approaches have in common that correctness is only guaranteed when analysis comes to an end. This is because the approximations used are under-approximations and thus the results are only guaranteed to be correct when a (global) fixed-point is reached.

However, in the **Manual** and **Automatic<sup>+</sup>** approaches the information in the global state is correct after any number of local analysis steps.

On the other hand, both the **Flattening** and **Automatic<sup>-</sup>** approaches are guaranteed to obtain the most accurate information possible, i.e., the least analysis graph, when a fixed-point is reached. In contrast, the **Manual** approach cannot guarantee optimal accuracy for two reasons. The first one is that there is no guarantee that modules will be processed the number of times that is necessary for an inter-modular fixed-point to be reached. Second, even if such a fixed-point is reached, it may not be the least fixed-point. This is because this approach uses over-approximations of the analysis results which are improved (“narrowed”) in the different analysis iterations until a fixed-point is reached. On the other hand, if there are no circular dependencies among predicates in different modules, then the fixed-point obtained will be the least one, i.e., the most accurate.

Regarding efficiency in time we will consider two cases. The first one is when we have to perform analysis of the program unit from scratch. In this case, **Flattening** can be highly optimized in order to converge quickly to a fixed-point. In contrast, in this situation the instances of the modular framework have the disadvantage that loading and unloading modules during analysis introduces a significant overhead. As a result, in order to maintain the number of context changes low, call patterns may be solicited from imported modules which use temporary information and which are not needed in the final analysis graph. These call patterns which end up being useless are known as *spurious* versions. This problem also occurs in **Flattening**, though to a much lesser degree if good algorithms are used. Therefore, the modular approaches may end up performing work which is speculative, and thus the total amount of work performed in the automatic approaches to modular analysis is in principle an upper bound of that needed in **Flattening**.

On the other hand, consider the second case in which a relatively large amount of intra-modular analysis has already taken place for the modules to be analyzed in our programming unit and that the global information is persistent. In this case, the automatic approaches can update their global data structures using the precomputed information, rather than starting from scratch as is done in **Flattening**. In such a case the automatic approaches may perform much less work than **Flattening**. It is to be expected that once module  $m$  becomes stable, i.e., it is fully developed, it will quickly be analyzed for a relatively large set of calling patterns. In such a case it is likely that it will be possible to analyze any other module  $m'$  which uses  $m$  by simply reusing the existing analysis results for  $m$ . This is specially true in the case of *library modules*, as discussed in Section 9.3.

Regarding the efficiency in terms of memory, it is to be expected that the instances of the modular framework will outperform the non-modular, flattening approach. This was in fact already observed in the case of [4]. Indeed, one important practical difficulty that appears during the (monolithic) analysis of large programs is that the amount of information which is kept in memory is very large and the storage needed can become too large to fit in memory. The

**Table 1.** Comparison of Approaches to Modular Analysis

	Flattening	Manual	Automatic <sup>+</sup>	Automatic <sup>-</sup>
Scheduling policy	automatic	manual	automatic	automatic
Success policy	$SP^-$	$SP^+$	$SP^+$	$SP^-$
Entry policy	top-level	all	top-level	top-level
Module-aware	no	yes	yes	yes
No Rean. of all CPs	no	n/a	yes	yes
Correct	at fixed-point	yes	yes	at fixed-point
Accurate	yes	no	no circularities	yes
Efficient in time	yes	n/a	no	no
Efficient in memory	no	yes	yes	yes
Termination	finite asc. chains	finite asc. chains	finite chains	finite asc. chains

modular framework proposed needs less memory because: a) at each point in time, only one module requires to be loaded in the code area, and b) the local answer table only needs to hold entries for the module being analyzed, and not for other modules. Also, in general, the total amount of memory required to store the global data structures is not very high when compared to the memory required locally for the different modules. In addition, not all the global data structures are required when analyzing a module  $m$ , but only that associated with the modules in  $related(m)$ .

Finally, regarding termination, except for **Flattening**, in which only one level of termination is required, the three other cases require two levels of termination: at the intra-modular and at the inter-modular level. In **Flattening**, since analysis results increase monotonically until a fixed-point is reached, termination is often guaranteed by considering description domains which do not contain infinite ascending chains: no matter what the current description is,  $top(\top)$ , which is trivially guaranteed to be a fixed-point, is only a finite number of steps away. Exactly the same condition is required for guaranteeing termination of **Automatic<sup>-</sup>**. The manual approach only requires guaranteeing intra-modular termination since the number of call patterns analyzed is finite. However, in the case **Automatic<sup>+</sup>**, finite ascending chains are required for ensuring local termination *and* finite descending chains are required for ensuring global termination. As a result, termination requires domains with finite chains, or appropriate widening operators.

In summary, the proposed two-level generic framework for analysis and its instantiations meet a good subset of our stated objectives. We hope the discussion and the concrete proposal presented in this paper will provide a better understanding of the handling of context-sensitive program analysis on modular programs and contribute to the widespread use of such context-sensitive analysis techniques for modular programs in practical systems. An implementation of the framework, as a generalization of the pre-existing CiaoPP modular analysis components, is currently being completed. In this context, we are experimenting with different scheduling policies for the global level, for concrete, practical analysis situations.

## Acknowledgments

This work was funded in part by projects ASAP (EU IST FET Programme Project Number IST-2001-38059) and CUBICO (MCYT TIC 2002-0055) and ARC IREX Grant X00106666. Part of this work was performed during a research stay of Germán Puebla and Jesús Correas at UNM supported by grants from the Secretaría de Estado de Educación y Universidades and by the Madrid Regional Government (CAM), respectively. Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM.

## References

1. F. Besson and T. Jensen. Modular class analysis with datalog. In *10th International Symposium on Static Analysis, SAS 2003*, number 2694 in LNCS. Springer, 2003.
2. A. Bossi, M. Gabbrieli, G. Levi, and M.C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122(1,2):3–47, 1994.
3. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.
4. F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.
5. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
6. M. Codish, S.K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proc. POPL'93*, 1993.
7. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
8. María J. García de la Banda, Bart Demoen, Kim Marriott, and Peter J. Stuckey. To the Gates of HAL: A HAL Tutorial. In *International Symposium on Functional and Logic Programming*, pages 47–66, 2002.
9. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.
10. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.
11. A. Kelly, A. Macdonald, K. Marriott, H. Søndergaard, and P.J. Stuckey. Optimizing compilation for CLP( $\mathcal{R}$ ). *ACM Transactions on Programming Languages and Systems*, 20(6):1223–1250, 1998.
12. K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

13. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
14. Nicholas Nethercote. The Analysis System of HAL. Master’s thesis, Monash University, 2002.
15. Christian W. Probst. Modular Control Flow Analysis for Libraries. In *Static Analysis Symposium, SAS’02*, volume 2477 of *LNCS*, pages 165–179. Springer-Verlag, 2002.
16. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in *LNCS*, pages 23–61. Springer-Verlag, September 2000.
17. G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in *LNCS*, pages 270–284. Springer-Verlag, September 1996.
18. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.
19. G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
20. G. Puebla and M. Hermenegildo. Abstract Specialization and its Applications. In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM’03)*, pages 29–43. ACM Press, June 2003. Invited talk.
21. A. Rountev, B.G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *ESEC/FSE’99*, volume 1687 of *LNCS*, pages 235–252. Springer-Verlag, 1999.
22. Y. M. Tang and P. Jouvelot. Separate abstract interpretation for control-flow analysis. In *Theoretical Aspects of Computer Software (TACS ’94)*, number 789 in *LNCS*. Springer, 1994.
23. W. Vanhoof and M. Bruynooghe. Towards modular binding-time analysis for first-order mercury. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.