3. A Generic Preprocessor for Program Validation and Debugging

Germán Puebla, Francisco Bueno, and Manuel Hermenegildo

School of Computer Science
Technical University of Madrid, Spain
email: {german,bueno,herme}@fi.upm.es

We present a generic preprocessor for combined static/dynamic validation and debugging of constraint logic programs. Passing programs through the preprocessor prior to execution allows detecting many bugs automatically. This is achieved by performing a repertoire of tests which range from simple syntactic checks to much more advanced checks based on static analysis of the program. Together with the program, the user may provide a series of assertions which trigger further automatic checking of the program. Such assertions are written using the assertion language presented in Chapter 2, which allows expressing a wide variety of properties. These properties extend beyond the predefined set which may be understandable by the available static analyzers and include properties defined by means of user programs. In addition to user-provided assertions, in each particular CLP system assertions may be available for predefined system predicates. Checking of both user-provided assertions and assertions for system predicates is attempted first at compile-time by comparing them with the results of static analysis. This may allow statically proving that the assertions hold (i.e., they are validated) or that they are violated (and thus bugs detected). User-provided assertions (or parts of assertions) which cannot be statically proved nor disproved are optionally translated into run-time tests. The implementation of the preprocessor is generic in that it can be easily customized to different CLP systems and dialects and in that it is designed to allow the integration of additional analyses in a simple way. We also report on two tools which are instances of the generic preprocessor: CiaoPP (for the Ciao Prolog system) and CHIPRE (for the CHIP CLP(FD) system). The currently existing analyses include types, modes, non-failure, determinacy, and computational cost, and can treat modules separately, performing incremental analysis.

3.1 Introduction

Constraint Logic Programming (CLP) [3.34] is a powerful programming paradigm which allows solving hard combinatorial problems. As (constraint) logic programming systems mature and further and larger applications are built, an increased need arises for advanced development and debugging environments. In the current state of the practice, the tasks of validation and debugging of CLP programs are very costly in the software development process. This is especially true when the problems to be solved involve a large number of variables, constraints, and states.

As discussed in previous chapters, such advanced environments will likely comprise a variety of co-existing tools ranging from declarative debuggers to execution visualizers, such as those presented in this book. See also [3.24, 3.23] for a discussion on possible debugging scenarios. In order to have a satisfactory program we need at least the following two properties:

- The program is *correct*. During the development phase we often have programs which produce *wrong results* and/or fail to produce results for some valid input data, i.e., *missing results*. Also, the program may generate *run-time errors*. Though it is true that these problems mainly occur in the first stages of development, they may also appear later, for example, if the program is modified (perhaps to improve performance). The approach we propose is to use the generic preprocessor presented in this chapter for such problems.
- The program is reasonably *efficient*. Though a program may be correct, it may also be the case that the program takes too long to terminate for practical purposes. If this occurs, the program may need to be modified. For example, the constraint order, modeling of the problem, the solver being used, the heuristics applied, etc. may need to be changed. Though the preprocessor may be useful for efficiency debugging, for example using cost analysis, other tools are also of use, for example the visualization tools presented in chapters 6 through 10.

Correctness checking, or checking for short, can be performed either at compile-time, i.e., before executing the program, or at run-time, i.e., while executing the program, or both. In both cases it is important that such checking be performed *automatically*. Also, the sooner in the development process an incorrect program is detected, the better. Thus, compile-time checking is generally preferable to run-time checking. In this chapter, we present a preprocessor for correctness checking which is automatic, generic (in the sense that it can be instantiated to different CLP languages), and it is based on a certain preference for compile-time checking over run-time checking. It is based on work previously presented in [3.39, 3.32, 3.31, 3.30].

3.1.1 Design of the Preprocessor

Most existing CLP systems perform correctness checks in one way or another. The classical scenario of such checking is depicted in Figure 3.1. Compile-time checking typically involves at least performing *syntactic checking*. Programs to be executed have to adhere to a given syntax. Those programs which do not satisfy such syntax are flagged at compile-time as *syntactically incorrect* instead of being executed, as depicted in Figure 3.1. Clearly, the fact that a program is syntactically correct does not guarantee that the program is

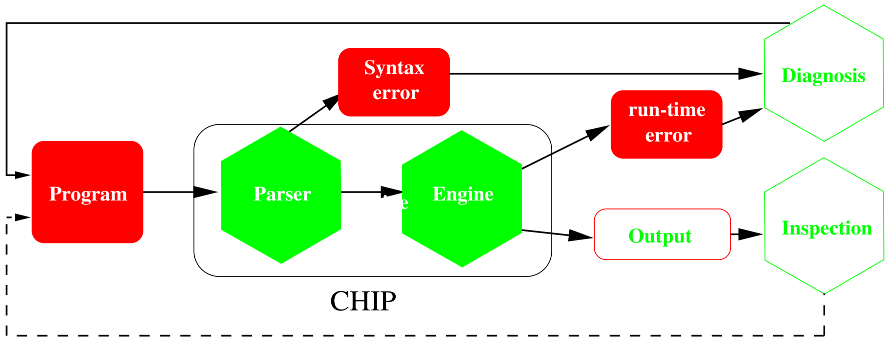

Fig. 3.1. Correctness Checking in CLP Systems.

correct. Run-time checking typically involves at least checking that builtin and library predicates are called as expected. If, for example, the types of the arguments are not those expected, the code for the predicate may not be valid and the possible results are not guaranteed to be correct. When some call to a builtin or library predicate is detected to be invalid, an error message is issued and generally execution aborted.

Example 3.1.1. Consider the following query and error message using CHIP:

```
22?- X=g,indomain(X).
Error 119 : domain variable expected in indomain(g)
```

where the CHIP builtin `indomain` assigns to its argument (`X` in this case) one of the values which remain available in its domain and enumerates all possible values on backtracking. However, this only makes sense if `X` is a finite domain variable. Clearly, this is not the case in the example, as `X` is bound to the constant `g`.

Unfortunately, the kinds of incorrectness errors which most existing CLP systems automatically detect either at compile-time or run-time are very limited. One reason for this is that the system does not have clear criteria to flag a program as incorrect other than syntax errors and invalid run-time calls to system predicates. This is because the system does not know the expected program behaviour which the user has in mind. A usual way to increase the criteria to detect incorrect programs is to provide the system with *assertions* (see Chapter 2) describing user's intentions, and which the system can check either at compile-time or at run-time.

The preprocessor we present greatly extends the checking capabilities of CLP systems in order to automatically detect more incorrectness symptoms. We have extended the traditional syntactic checks and also incorporated assertion-based checking both at compile-time and run-time. In our preprocessor, compile-time checking is based on a range of powerful program an-

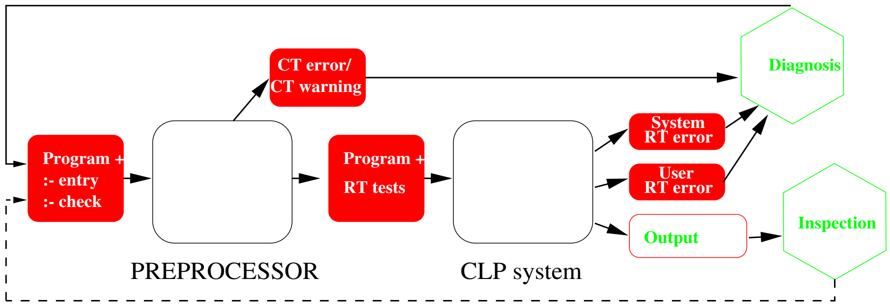

Fig. 3.2. Using the Preprocessor.

alyzers, which gives the system a new flavour: sometimes (semantic) errors can be detected even without users having provided any assertion at all.

We call our system a *preprocessor* because programs are expected to be passed through it prior to being run on the CLP system. When a program is passed through the preprocessor, i.e., *preprocessed*, two kinds of messages may be produced: *error* messages and *warning* messages (depicted in Figure 3.2 as `CT error` and `CT warning`, respectively). Error messages are issued when evidence has been found that the program is definitely incorrect (i.e., an incorrectness symptom has been found). Whenever an error message is issued the program should be corrected, rather than executed. Warning messages are issued when the program is suspect of being incorrect. Thus, warning messages do not always mean that the program is incorrect, but they often help detecting bugs. Warning messages should also be taken into account in order to decide whether they correspond to bugs or not. If they do not, they can often be avoided by making slight modifications to the program. Thus, the next time the program is preprocessed such warning messages will not be issued any more. This is not strictly required, but avoids distracting messages which do not correspond to bugs. If messages are issued and the program modified, it should be preprocessed once again. Several iterations may be required until no more errors (nor warnings) are flagged.

As depicted in Figure 3.2, the starting point for assertion-based correctness debugging is a set of `check` assertions (see Chapter 2) which provide a (partial) description of the intended behaviour of the program. Besides `check` assertions, the programmer may also supply `entry` assertions for the program, which describe the valid queries to the program.

A fundamental design principle in our preprocessor is to be as unrestrictive as possible with regards to what the user needs to provide. To start with, and as mentioned in Chapter 2, the assertion language allows expressing properties which are much more general than, for example, traditional type declarations, and such that it may be undecidable whether they hold or not for a given program. Once we lift the requirement that assertions be decidable, it is also natural to allow assertions to be *optional*: specifications may be given only for some parts of the program and even for those parts the information given may be incomplete. This allows preprocessing existing programs without having to add assertions to them and still being able to detect errors. And if we decide to add some assertions to our programs, such assertions may be given for only some procedures or program points, and for a given predicate we may perhaps have the type of one argument, the mode of another, and no information on other arguments. Finally, we would like the system to be able to *generate* assertions which describe the behaviour of the existing program, especially for parts of the program for which there are no `check` assertions. These assertions will have the status `true` (see Chapter 2) and can be visually inspected by the user for checking correctness.[1]

Note that, although neither `entry` nor `check` assertions are strictly required, the more effort the user invests in providing accurate ones, the more bugs can be automatically detected. Also, since `check` assertions may not encode a complete specification of the program, the fact that all `check` assertions hold does not necessarily mean that the program is correct, i.e., that it behaves according to the user's intention. It may be the case that the program satisfies the existing `check` assertions but violates some part of the specification which the user has decided not to provide. However, for the purposes of assertion checking, we say that a program is *correct* w.r.t. the given assertions for given valid queries if all its assertions have been proved for all the states that may appear in the computation of the program with the given valid queries.

A consequence of our assumptions so far, is that the overall framework needs to deal throughout with *approximations* [3.7, 3.17, 3.31]. Thus, while the system can be complete with respect to decidable properties (e.g., certain type systems), it cannot be complete in general, and the system may or may not be able to prove in general that a given assertion holds. The overall operation of the system will be sometimes imprecise but must always be *safe*. This means that all violations of assertions flagged by the preprocessor should indeed be violations, but perhaps there are assertions which the system cannot determine to hold or not. As already discussed in Chapter 2, this also means that we cannot in general reject a program because the preprocessor has not been able to prove that the complete specification holds.

Thus, and returning to our overall debugging process using the preprocessor (Figure 3.2), it may be the case that after some iterations of the checking/program correction cycle the preprocessor is not capable of detecting additional errors at compile-time nor to guarantee that the program is correct w.r.t. the existing `check` assertions. In such a situation the preprocessor allows the possibility of performing dynamic checking of assertions, i.e., introducing run-time tests into the program (indicated as `Program + RT tests`

[1] Note however that if `check` assertions exist for such parts of the program they are automatically checked.

in Figure 3.2). Execution of the resulting program on the underlying CLP system may then issue two kinds of error messages: those directly generated by the CLP system (`System RT error`) due to incorrect run-time calls to system predicates and those produced by the code added for run-time checking (`User RT error`) if some user-provided assertion is detected to be violated while executing the program.[2]

Our approach is strongly motivated by the availability of powerful and mature static analyzers for (constraint) logic programs, generally based on abstract interpretation [3.17]. These analyzers have proved quite effective in statically *inferring* a wide range of program properties accurately and efficiently, for realistic programs, and with little user input (see, e.g., [3.33, 3.38, 3.13, 3.26, 3.27, 3.36, 3.5, 3.6] and their references). Such properties can range from types and modes to determinacy, non-failure, computational cost, independence, or termination, to name a few. Traditionally, the results of static analyses have been applied primarily to program optimization: parallelization, partial evaluation, low-level code optimization, etc. However, as we have seen, herein we will be applying static analysis in the context of *program development* (see, e.g., [3.2, 3.7, 3.31, 3.30]), and, in particular, in validation and error detection. This fits within a larger overall objective (achieved to a large extent in CiaoPP [3.30]), which is to combine both program optimization and debugging into an integrated tool which uses multiple program analyses and (abstract) program specialization [3.41, 3.40] as the two main underlying techniques.

3.1.2 Chapter Outline

In the following sections, we describe the preprocessor, mainly by means of examples, and discuss some technical issues of the correctness checking it performs. We present the overall framework of the preprocessor (i.e., we give some insight into the blank box of Figure 3.2) in Section 3.2. In Section 3.3 the techniques used for assertion checking both at compile-time and at run-time are discussed. The ideas presented are summarized and exemplified in Section 3.6 following a running example. In Section 3.5 we present how to customize the preprocessor for a particular CLP system. Finally, Section 3.7 discusses some practical hints on the use of assertions for correctness checking in our system.

In the rest of the text, we will use examples written in both CHIP [3.16, 3.1] and Ciao[3] [3.4, 3.29] and output from the corresponding preprocessors, i.e., CHIPRE [3.9] and CiaoPP [3.8, 3.30]. We will be mixing explanations at a tutorial level with some more technical parts in an effort to provide material

[2] As we will see, the messages from system predicates can in fact be also made come from the assertions present in the system libraries or those used to describe the built-ins.

[3] The Ciao system is available at `http://www.clip.dia.fi.upm.es/Software/`.

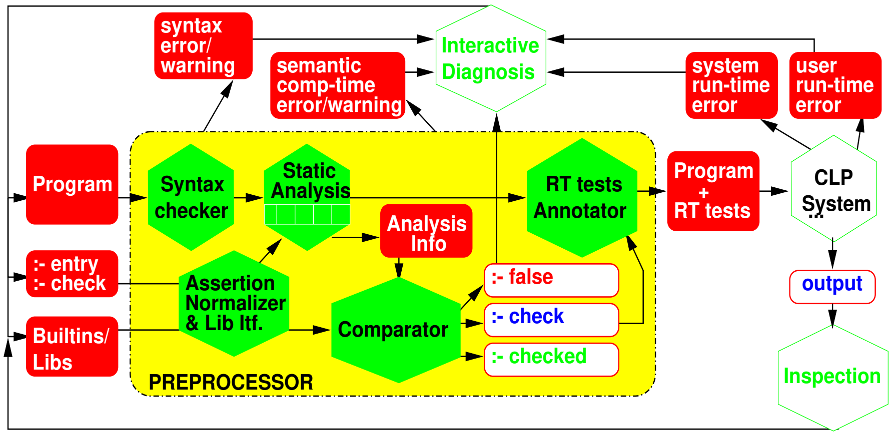

Fig. 3.3. Architecture of the Preprocessor

that is instructive for newcomers but also has enough detail for more expert readers.

3.2 Architecture and Operation of the Preprocessor

The preprocessor is a complex system composed of several tools. Figure 3.3 depicts the overall architecture of the generic preprocessor. Hexagons represent the different tools involved and arrows indicate the communication paths among the different tools. It is a design objective of the preprocessor that most of such communication be performed also in terms of assertions. This has the advantage that at any point in the debugging process the information is easily readable by the user. In this section we provide an overall description of the different components of the preprocessor and give an overview of the kinds of bugs the preprocessor can automatically detect by means of examples. Further examples and details on some components will be given in other sections throughout the remainder of the chapter.

3.2.1 The Syntax Checker

As mentioned before, the preprocessor performs an extension of the syntax-level checking performed on programs prior to execution by traditional CLP systems. Though for simplicity only error messages were depicted in Figure 3.1, the preprocessor (as the CLP system) may issue error and/or warning messages (Figure 3.3).

Example 3.2.1. Consider the program in Figure 3.4, which contains a tentative version of a CHIP program for solving the *ship* scheduling problem, which

```
solve(Upper,Last,N,Dis,Mis,L,Sis):-
        length(Sis, N),
        Sis :: 0..Last,
        Limit :: 0..Upper,
        End :: 0..Last,
        set_precedences(L, Sis, Dis),
        cumulative(Sis, Dis, Mis, unused, unsed, Limit, End, unused),
        min_max(labeling(Sis), End).

labeling([]).
labeling([H|T]):-
        delete(X,[H|T],R,0,most_constrained),
        indomain(X),
        labeling(R).

set_precedences(L, Sis, Dis):-
        Array_starts=..[starts|Sis], % starts(S1,S2,S3,...)
        Array_durations=..[durations|Dis], % durations(D1,D2,D3,...)
        initialize_prec(L,Array_starts),
        set_pre_lp(l, array_starts, Array_durations).

set_pre_lp([],Array_starts).
set_pre_lp([After #>= Before|R], Array_starts, Array_durations):-
        arg(After, Array_starts, S2),
        arg(Before, Array_starts, S1),
        arh(Before, Array_durations, D1),
        S2 #>= S12 + D1,
        set_pre_lp(R, Array_starts, array_durations).

initialize_prec(_,_).
```

Fig. 3.4. A tentative *ship* program in CHIP

is one of the typical benchmarks of CHIP. When preprocessing this program with CHIPRE, i.e., an implemented instance of the generic preprocessor for the CHIP system, the following messages are generated:

```
WARNING: (lns 34-35) predicate set_pre_lp/2:
                     has singleton variable(s) [Array_starts]
WARNING: (lns 35-42) predicate set_pre_lp/3:
                     already defined with arity 2
WARNING: (lns 35-42) predicate set_pre_lp/3:
                     has singleton variable(s) [S1,S12]
WARNING: (lns 35-42) predicate arh/3 undefined in source
```

The first message indicates that the variable `Array_starts` is a *singleton*, i.e., it only appears once in the clause. This often indicates that we have mistyped the name of a variable. However, in this case this does not correspond to a bug. `Array_starts` is a singleton because its value is not used anywhere else in the clause. The second message is issued because there are two predicates in the program with the same name but different arity. This often indicates that we have forgotten an argument in the head of a clause or we have added extra ones. In fact, we have forgotten the third argument in the first clause of `set_pre_lp`. The third message marks both `S1` and `S12` as singletons. This is actually a bug as we have typed `S12` instead of `S1` in the second clause. The last message indicates that there is a call to a predicate which is not defined. This is because in the second clause for predicate `set_pre_lp` there is a call `arh(Before, Array_durations, D1)`, where the name of the CHIP builtin predicate `arg` has been mistyped. However, unless we actually check that `arh` does not correspond to some user-defined predicate (by looking at the whole program) we cannot be sure that it corresponds to an undefined predicate.[4]

In addition to warning messages like the ones seen in the example above, the preprocessor also issues warning messages when the clauses which define a predicate are discontiguous. This may indicate that we have mistyped the name of the predicate in a clause or that an old clause for the predicate which should have been deleted is still in the program text. Some CLP systems already perform some of the checks discussed, mainly the singleton variable check.

As mentioned before, it is usually easy to make small (style) modifications to a program in order to avoid generating warning messages which do not correspond to bugs. Singleton variables can be replaced by *anonymous variables*, whose name start with an underscore. Predicates with the same name but different arities can be renamed to avoid name coincidence. Discontiguous clauses can always be put together. However, if we prefer not to modify the program, we can also instruct the preprocessor not to issue warning messages by setting the corresponding flag off.

Example 3.2.2. The following directive tells the preprocessor not to issue any warning message for predicates with the same name and different arity (until the flag is set to on):

```
:- set_prolog_flag(multi_arity_warnings,off).
```

Guided by the warning messages discussed above, we now replace the definition of predicate `set_pre_lp` by the following one, for which no syntactic warning message is issued anymore:

```
set_pre_lp([],_,_).
set_pre_lp([After#>=Before|R],Array_starts,Array_durations):-
        arg(After, Array_starts, S2),
        arg(Before, Array_starts, S1),
        arg(Before, Array_durations, D1),
        S2 #>= S1 + D1,
        set_pre_lp(R, Array_starts, array_durations).
```

[4] A well designed module system is instrumental in the task of detecting undefined predicates, especially when performing separate compilation [3.12].

The extra syntactic checking done by the preprocessor is based on that performed by the Ciao modular compiler [3.11]. This checking is simple to implement, efficient to perform, as it does not need any complex analysis, and very relevant in practice: it allows early bug detection and it does not require any additional input from the user.

3.2.2 The Static Analyzers

Though syntactic checking is simple, it is also very limited. In fact, there are plenty of incorrect programs which syntactically look fine. In order to go beyond syntactic checking, we require *semantic checking*. This kind of checking involves "understanding" to some extent what a piece of code does. This task is performed at compile-time by so-called *static analysis*. Static analysis is capable of inferring some useful properties of the behaviour of a program without actually running it. By running the program on sample input data we can obtain very precise information on the behaviour of the program, but such information is not guaranteed to be correct for other input data. In contrast, information obtained by means of static analysis is indeed guaranteed to be correct for *any* input data. However, analysis has to approximate its results in order to ensure termination, and this causes analysis to lose some information.

Example 3.2.3. Consider the following toy program:

```
p(X,Y):- Y is 2*X + 1.
```

By executing the program with input value `X=0` we can conclude that on success of the program `Y=1`. Static analysis can conclude that, for example, on success of the program the following properties hold for argument `Y`: `integer(Y)` (using type analysis), `ground(Y)` (using mode analysis), and `odd(Y)` (using parity analysis) independently of the particular value of `X`.

Unfortunately, static analysis is a hard task. In fact, no existing commercial CLP system contains a full fledged static analyzer.[5] Thus, they cannot perform effective semantic compile-time checking. However, several generic analysis engines, generally based on abstract interpretation [3.17], such as PLAI [3.38], GAIA [3.13] and the CLP($\mathcal{R}$) analyzer [3.37], facilitate construction of static analyzers for (C)LP. These generic engines have the *description domain* as parameter. Different domains give analyzers which provide different kinds of information and degrees of accuracy.

[5] Some exceptions do exist in the academic world, notably the &-Prolog system (the predecessor of Ciao and the first LP system to include a global analyzer –PLAI– to perform optimization tasks), the Aquarius and PARMA systems (the first LP systems to perform low-level optimizations based on global analysis), and the latest version of the CLP(R) compiler.

Static analysis can be *local* or *global*. In local analysis different parts of the program can be treated independently from others, for example, by processing one clause at a time. In global analysis, the whole program has to be taken into account as the information obtained when processing other clauses (predicates) is possibly needed for processing a given clause (predicate). Thus, global analysis usually requires several iterations of analysis over the program code. Also, care must be taken to ensure termination of analysis. As a result, local analysis is generally simpler and more efficient than global analysis, but also less accurate.

The success of compile-time checking greatly depends on the accuracy of static analysis. As mentioned in Chapter 2, if analysis is goal-dependent, the accuracy of analysis can be improved by providing accurate `entry` declarations.

Example 3.2.4. In the ship program, all initial queries to the program should be to the `solve` predicate. However, the compiler has no way to automatically determine this. Thus, if no `entry` assertions are given for the program then most general entry assertions, i.e., of the form '`:- entry p(X1,...,Xn) : true.`' have to be assumed for *all* predicates `p/n` in the program.[6] However, if some entry assertion(s) exist they are assumed to cover all possible initial calls to the program. Thus, even the simplest entry declaration which can be given for predicate `solve`, i.e., '`:- entry solve(A,B,C,D,E,F,G) : true.`' (which can also be written using some syntactic sugar as '`:- entry solve/7.`') is very useful for goal-dependent static analysis. Since it is the only `entry` assertion, the only calls to the rest of the predicates in the program are those generated during computations of `solve/7`. This allows analysis to start from the predicate `solve/7` only, instead of from all predicates, which can result in increased precision. However, analysis will still make no assumptions regarding the arguments of the calls to `solve/7`. This can be improved using a more accurate entry declaration such as the following:

```
:- entry solve/7 :
         int * int * int * list(int) * list(int) * list * term.
```

which is syntactic sugar for '`:- entry solve(A,B,C,D,E,F,G): (int(A),int(B), int(C),list(D,int),list(E,int),list(F),term(G)).`'. It gives the types of the input arguments, and describes more precisely the valid input data.[7]

[6] Another advantage of a strict module system such as that of Ciao [3.12] is that only exported predicates of a module can be subject to initial queries. Thus most general `entry` assertions need only be assumed for exported predicates.

[7] Note that since, as mentioned in Chapter 2, properties (and, therefore, types) are considered *instantiation properties* by default, the assertion above also specifies a *mode*: all arguments except the last two are required to be ground on calls. The last but one argument is only *required* to be instantiated to a list skeleton, while no constraint is placed on the last argument.

3.2.3 Consistency of the Analysis Results

The results of static analysis are often good indicators of bugs, even if no assertion is given. This is because "strange" results often correspond to bugs. As is the case with the syntactic warnings presented before, these indicators should be taken with care (as they do not ensure any violation of assertions) and thus warning messages rather than error messages are produced. An important observation is that plenty of static analyses, such as modes and regular types, compute over-approximations of the success sets of predicates. Then, if such an over-approximation corresponds to the empty set then this implies that such predicate never succeeds. Thus, unless the predicate is dead-code,[8] this often indicates that the code for the predicate is erroneous since every call either fails finitely (or raises an error) or loops.

Example 3.2.5. When preprocessing the current version of our example program using *regular types* [3.44, 3.18, 3.26] (see also Chapter 4 for a detailed discussion on regular types) analysis we get the following messages:

```
WARNING: Literal set_precedences(L, Sis, Dis)
            at solve/7/1/5 does not succeed!
WARNING: Literal set_pre_lp(l, array_starts, Array_duration)
            at set_precedences/3/1/4 does not succeed!
```

The first warning message refers to a literal (in particular, the 5th literal in the 1st clause of `solve/7`) which calls the predicate `set_precedences/3`, whose success type is empty. Also, even if the success type of a predicate is not empty, i.e., there may be some calls which succeed, it is possible to detect that at a certain program point the given call to the predicate cannot succeed because the type of the particular call is incompatible with the success type of the predicate. This is the reason for the second warning message.[9] Note that the predicate `set_pre_lp/3` can only succeed if the value at the first argument is compatible with a list. However, the call `set_pre_lp(l, array_starts, Array_duration)` has the constant `l` at the first argument. This is actually a bug, as `l` should be `L`. Once we correct this bug, in subsequent preprocessing of the program both warning messages disappear. In fact, the first one was also a consequence of the same bug which propagated to the calling predicates of `set_precedences/3`.

3.2.4 The Assertion Normalizer

As seen in Chapter 2, the assertion language in addition to the "basic" syntax for assertions also has an "extended" syntax which can be seen as syntactic

[8] If analysis is goal-dependent and thus also computes an over-approximation of the calling states to the predicate, predicates which are dead-code can often be identified by having an over-approximation of the calling states which corresponds to the empty set.

[9] This kind of reasoning can only be done if the static analysis used infers properties which are *downwards closed*.

sugar for writing assertions. The role of this module is to convert assertions possibly written in the extended syntax into the basic assertion language. For example, compound assertions (see Chapter 2) are converted into basic ones prior to compile-time checking. This module is represented in Figure 3.3 by the hexagon labeled `Assertion Normalizer & Lib Itf`. This module is also in charge of generating and reading *interface* files for modules. Interface files contain assertions describing the predicates exported by the module. This allows correctly analyzing and checking programs composed of several modules without having to preprocess the auxiliary modules over and over again [3.42]. I.e., interface files allow modular analysis and assertion checking.

3.2.5 The Assertion Comparator

A simple (but tedious) possibility in order to use the information obtained by static analysis for detecting correctness problems at compile-time is to visually inspect the analysis results: unexpected results often indicate correctness problems. A more attractive alternative is to automatically compare the analysis results with our expectations, given as assertions.

As depicted in Figure 3.3, this is done in the preprocessor by the tool named `Comparator`. The result might be that the assertion is validated or that it is proved not to hold. In the first case the corresponding assertions are rewritten as `checked` assertions; in the second case *abstract* symptoms are detected, the corresponding assertions are rewritten as `false` assertions, and error messages are presented to the user. It is also possible that an assertion cannot be proved nor disproved. In this case some assertions remain in `check` status, and warning messages could be presented to the user to indicate this. In the case that errors are generated, diagnosis should be started. One option is to use the type-based diagnoser presented in Chapter 4 if the properties in the assertion are regular types, or other forms of abstract diagnosis [3.14, 3.15], in order to detect the cause of the error. Also, as we will see, the preprocessor does perform a certain amount of diagnosis, in the sense that it locates not only the assertion from which the the error or warning is generated but also, for example, the clause body literal that originates the call which is identified to be in error (see Section 3.2.7). Also, we believe it is not difficult to extend the preprocessor to perform more extensive diagnosis, using the quite detailed information on dependencies between program points that the analyzers keep track of.

3.2.6 Assertions for System Predicates

A very important feature of the preprocessor, explained in detail in Section 3.5.1, is that the behaviour of system predicates (not only of user predicates) is given in terms of assertions. By *system predicates* we denote both builtin and library predicates provided by the programming system. As many as possible of these system predicates should be described using assertions. Such assertions for system predicates, which are depicted in Figure 3.3 as `Builtin/Libs`, are in principle meant to be written by system developers when generating a particular instance of the preprocessor for a given CLP system. For example, the assertions describing the system predicates in Ciao have been written by the authors of this chapter and other Ciao developers and are part of the Ciao system libraries, whereas the assertions describing the builtin predicates in CHIP have been written at COSYTEC (i.e., the CHIP developers).

The existence of assertions which describe system predicates is beneficial for at least two reasons. One which is seemingly simple but quite relevant in practice is the possibility of automatic generation of documentation (primarily reference manuals) directly from the assertions [3.28]. In fact, the documentation of the preprocessor itself [3.9, 3.8] is generated this way. Another one is that though system predicates are in principle considered correct under the assumption that they are called with valid input data, it is still of use to check that they are indeed called with valid input data. In fact, existing CLP systems perform this checking at run-time. The existence of such assertions allows checking the calls to system predicates at compile-time in addition to run-time in CLP systems which originally do not perform compile-time checking. This may allow automatically detecting many bugs at compile-time without any user-provided assertions. The only burden on the user is (1) to optionally provide one or more `entry` assertions which describe the valid queries and (2) to wait the time required by the preprocessor in order to both perform static analysis and compare the analysis results with the expected calls to system predicates.

Example 3.2.6. Consider the current version of the *ship* program, and assume that the only existing entry declaration is '`:- entry solve/7.`'. When preprocessing the program the following messages are issued:

```
ERROR: Builtin predicate
        cumulative(Sis,Dis,Mis,unused,unsed,Limit,End,unused)
        at solve/7/1/6 is not called as expected (argument 5):
         Called:    ^unsed
         Expected:  intlist_or_unused

ERROR: Builtin predicate arg(After,Array_starts,S2)
        at set_pre_lp/3/2/1 is not called as expected (argument 2):
         Called:    ^array_starts
         Expected:  struct
```

In error messages, the marker ^ is used to distinguish constants (terms) from regular types (which represent sets of terms). By default, values represent regular types. However, if they are marked with ^ they represent constants.

In our example, `intlist_or_unused` is a type since it is not marked with `^` whereas `^unsed` is a constant.[10]

The first message is due to the fact that the constant `unused` has been mistakingly typed as `unsed` in the fifth argument of the call to the CHIP builtin predicate `cumulative/8`. As indicated in the error message, this predicate requires the fifth argument to be of type `intlist_or_unused` which was defined when writing assertions for the system predicates in CHIP and which indicates that such argument must be either the constant `unused` or a list of integers. The exact definition of this type can be found in Section 3.5.1.

In the second message we have detected that we call the CHIP builtin predicate `arg/3` with the second argument bound to `array_starts` which is a constant (as indicated by the marker `^`) and thus of arity zero. This is incompatible with the expected call type `struct`, i.e., a structure with arity strictly greater than zero. In the current version of CHIP, this will generate a run-time error, whereas in other systems such as Ciao and SICStus, this call would fail but would not raise an error. Though we know the program is incorrect, the literal where the error is flagged, `arg(After, Array_starts, S2)` is apparently correct. We correct the first error and leave detection of the cause for the second error for later.

The different behaviour of seemingly identical builtin predicates (such as `arg/3` in the example above) in different systems further emphasizes the benefits of describing builtin predicates by means of assertions. They can be used for easily customizing static analysis for different systems, as assertions are easier to understand by naive users of the analysis than the hardwired internal representation used in ad-hoc analyzers for a particular system.

Run-time checking of assertions describing system predicates presents some peculiarities. When the system predicates (of a particular CLP system) already perform the required run-time checks, the preprocessor does not introduce any extra code for run-time checking, even if this option has been selected (in contrast to what happens for user-provided assertions). However, if we design a CLP system from the start which is always going to be used in conjunction with the preprocessor a very interesting alternative exists: we can avoid introducing in the system predicates any code for checking the calls, as that can be done by the preprocessor. The advantage of this approach is that programs can be more efficient, as the preprocessor may prove that many of the run-time tests are redundant, and thus not introduce run-time tests for them.

[10] Though it is always possible to define a regular type which contains a single constant such as `unsed`, we introduce the marker `^` ("quote") to improve the readability and conciseness of the messages. Note that defining such type explicitly instead would require inventing a new name for it and providing the definition of the type together with the error message.

3.2.7 Assertions for User-Defined Predicates

Up to now we have seen that the preprocessor is capable of issuing a good number of error and warning messages even if the user does not provide any `check` assertions. We believe that this is very important in practice. However, adding assertions to programs can be useful for several reasons. One is that they allow further semantic checking of the programs, since the assertions provided encode a partial specification of the user's intention, against which the analysis results can be compared. Another one is that they also allow a form of diagnosis of the error symptoms detected, so that in some cases it is possible to automatically locate the program construct responsible for the error. This is important since it is often the case that the preprocessor issues an error message at a program point which is actually far from the cause of the error. This happens, for example, when a predicate is called in the wrong way. Thus, one possibility is to visually inspect the program in order to detect which is (are) the wrong call(s) to the predicate among all the existing ones in the program. This does not seem like a good idea if the program being debugged is large and such predicate is used in several places. Thus, a better alternative is to add to the program an assertion on the calls to such predicate. This assertion can then be used by the preprocessor in order to automatically detect the wrong call(s) to the predicate.

Example 3.2.7. Consider again the pending error message from the previous iteration over the ship program. We know that the program is incorrect because (global) type analysis tells us that the variable `Array_starts` will be bound at run-time to the constant `array_starts`. However, by just looking at the definition of predicate `set_pre_lp` it is not clear where this constant comes from. This is because the cause of this problem is not in the definition of `set_pre_lp` but rather in that the predicate is being used incorrectly (i.e., its precondition is violated). We thus introduce the following `calls` assertion:

```
:- calls set_pre_lp(A,B,C): (struct(B),struct(C)).
```

In this assertion we require that both the second and third parameters of the predicate, i.e., `B` and `C` are structures with arity greater than zero, since in the program we are going to access the arguments in the structure of `B` and `C` with the builtin predicate `arg/3`.

The next time our ship program is preprocessed, having added the `calls` assertion, besides the pending error message of Example 3.2.6, we also get the following one:

```
ERROR: false assertion at set_precedences/3/1/4
       unexpected call (argument 2):
         Called:   ^array_starts
         Expected: struct
```

This message tells us the exact location of the bug, the fourth literal of the first clause for predicate `set_precedences/3`. This is because we have typed the constant `array_starts` instead of the variable `Array_starts` in such literal.

Thus, as shown in the example above, user-provided `check` assertions may help in locating the actual cause for an error. Also, as already mentioned, and maybe more obvious, user-provided assertions may allow detecting errors which are not easy to detect automatically otherwise.

Example 3.2.8. After correcting the bug located in the previous example, preprocessing the program once again produces the following error message:

```
ERROR: false assertion at set_pre_lp/3/2/5
       unexpected call (argument 3):
         Called:   ^array_durations
         Expected: struct
```

which would not be automatically detected by the preprocessor without user-provided assertions. The obvious correction is to replace `array_durations` in the recursive call to `set_pre_lp` in its second clause with `Array_durations`. After correcting this bug, preprocessing the program with the given assertions does not generate any more messages.

Finally, and as already discussed in Section 3.1.1, if some part of an assertion for a user-defined predicate has not been proved nor disproved during compile-time checking, it can be checked at run-time in the classical way, i.e., run-time tests are added to the program which encode in some way the given assertions. Introducing run-time tests by hand into a program is a tedious task and may introduce additional bugs in the program. In the preprocessor, this is performed by the `RT tests Annotator` tool in Figure 3.3. How run-time checks are introduced is discussed in detail in Section 3.4.

3.3 Compile-Time Checking

Compile-time checking of assertions is conceptually more powerful than run-time checking. However, it is also more complex. Since the results of compile-time checking are valid for *any* query which satisfies the existing `entry` declarations, compile-time checking can be used both to detect that an assertion is violated and to prove that an assertion holds for any valid query, i.e., the assertion is validated. The main problem with compile-time checking is that it requires the existence of suitable static analyses which are capable of proving the properties of interest. Unfortunately, static analyzers are complex to build and also some properties are very difficult to prove without actually running the program. On the other hand, though run-time checking is simpler to implement than compile-time checking, it also has important drawbacks. First, it requires test cases, i.e., sample input data, which typically have an incomplete *coverage* of the program execution paths. Therefore, run-time checking cannot be used in general for proving that a program is correct with respect to an assertion, i.e., that the assertion is `checked`, as this would require testing the program with all possible input values, which is in general unrealistic. Second, run-time checking clearly introduces overhead into program execution. Thus, it is important that we have the possibility of turning it on and off, as is the case in the preprocessor.

We now show informally how the actual checking of the assertions at compile-time is performed by means of an example. Then, we briefly discuss on the technique used in the preprocessor for "reducing" (i.e., validating and detecting violations of) assertions. Precise details on how to reduce assertions at compile-time can be found in [3.39].

Example 3.3.1. Assume that we have the following declarations of properties and user-provided assertions:

```
:- shfr prop ground/1.
:- shfr prop var/1.
:- regtype prop list/2.
list([],_P).
list([X|Xs],P):- P(X), list(Xs,P).
:- non_failure cprop does_not_fail/1.
:- cprop terminates/1.

:- check success p(X,Y) : ground(X) =>  ground(Y).
:- check success p(X,Y) =>  (list(X,int), list(Y,int)).
:- check comp    p(X,Y) :
                (list(X,int), var(Y)) + (does_not_fail,terminates).
```

which declare `ground/1`, `var/1`, and `list/2` as properties of execution states and `does_not_fail/1` and `terminates/1` as properties of computations (see Chapter 2). In addition, the declarations inform the preprocessor that properties `ground/1` and `var/1` can be treated using the inference system `shfr`, property `list/2` using `regtypes`, and `does_not_fail/1` using the `non_failure` inference system.[11]

As already mentioned in Chapter 2, assertion checking can be seen as computing the truth value of assertions by composing the value $eval(AF, \theta, P, IS)$ of the atomic properties AF at the corresponding stores θ reachable during execution. In the case of compile-time checking we must consider all possible stores reachable from any valid query. The abstract interpretation-based inference systems `shfr` and `regtype` compute a description (abstract substitution) for the calls and success states of each predicate (in fact they also

[11] It is worth mentioning that since the properties shown in the example above are quite standard it is good practice to have their declarations (and possibly also their definitions) in a library module, so that users can simply include the module in their code rather than having to write the declarations from scratch for every new program. Also, in each particular instantiation of the preprocessor a set of such library modules should exist which are adapted to the analyses available.

do so for every program point). In compile-time checking we consider an *abstract evaluation* function $eval_\alpha(AF, \lambda, P, IS)$ in which the concrete store θ has been replaced by an abstract description λ. We denote by $\gamma(\lambda)$ the set of stores which a description λ represents. Correctness of abstract interpretation guarantees that $\gamma(\lambda)$ is a safe approximation of the set of all possible stores reached from valid initial queries, i.e., all such states are in $\gamma(\lambda)$.

Example 3.3.2. After performing static analysis of the program (whose text we do not show as the discussion is independent of it) using `shfr` and `regtypes` we obtain a description of the calls and success states for predicate `p/2`. For readability, we now show the results of such analyses in terms of assertions:

```
:- true success p(X,Y):(ground(X),var(Y)) => (ground(X),ground(Y)).
:- true success p(X,Y):(list(X,int),term(Y))=>(list(X,int),int(Y)).
```

We denote by $\lambda_c(\texttt{p/2})$ and $\lambda_s(\texttt{p/2})$ the description of the calling and success states, respectively, of `p/2`. In our example, the static inference system `shfr` allows us to conclude that the evaluation of the three atomic properties $eval_\alpha(\texttt{ground(X)}, \lambda_c(\texttt{p/2}), P, \texttt{shfr})$, $eval_\alpha(\texttt{ground(Y)}, \lambda_s(\texttt{p/2}), P, \texttt{shfr})$, and $eval_\alpha(\texttt{var(Y)}, \lambda_c(\texttt{p/2}), P, \texttt{shfr})$ take the value *true*. Additionally, the `regtypes` analysis determines that on success of `p/2`, i.e., in $\lambda_s(\texttt{p/2})$, the type of argument `Y` is `int`, which is a predefined type in `regtypes`. This type is incompatible with `Y` being a list of integers, which is what was expected. Thus, $eval_\alpha(\texttt{list(Y,int)}, \lambda_c(\texttt{p/2}), P, \texttt{regtypes})$ takes the value $false$. The implementation of $eval_\alpha$ in the preprocessor is discussed below and is based on the notion of *abstract executability* [3.41, 3.40].

Regarding the `non_failure` inference system, we assume it is implemented (as in [3.19]) as a program analysis which uses the results of the `shfr` and `regtypes` analyses for approximating the calling patterns to each predicate, and then infers whether the program predicates with the given calling patterns might fail or not based on whether the type is recursively "covered".

Example 3.3.3. Assume that `non_failure` analysis concludes that `p/2` does not fail for its calling pattern `p(X,Y):(list(X,int),var(Y))`. This can be indicated by an assertion of the form:

```
:- true comp p(X,Y) : (list(X,int), var(Y)) + does_not_fail.
```

and thus $eval_\alpha(\texttt{does\_not\_fail(p(X,Y))}, \lambda_c(\texttt{p/2}), P, \texttt{non\_failure})$ is guaranteed to take the value *true*.

Assume also that we do not have any static inference system which can decide whether `terminates(p(X,Y))` with calling pattern `(list(X,int),var(Y))` holds or not. Thus, $eval_\alpha$ for this property has to take the value "don't know".[12]

[12] Note that this could also happen even if a termination analysis exists in the system. Termination analyses may not able in general to determine whether a given computation terminates or not.

The next step consists of composing and simplifying the truth value of each logic formula from the truth value computed by $eval_\alpha$.

Example 3.3.4. After composing the results of evaluating each atomic property in the assertion formulae we obtain:

```
:- check success p(X,Y) : true =>  true.
:- check success p(X,Y) =>  (true, false).
:- check comp    p(X,Y) : (true, true) +   (true, terminates).
```

We can now apply typical simplification of logical expressions and obtain:

```
:- check success p(X,Y) : true =>  true.
:- check success p(X,Y) =>  false.
:- check comp    p(X,Y) : true +   terminates.
```

The third and last step is to obtain, if possible, the truth value of the assertion as a whole. As assertion takes the value *true*, i.e., it is validated if either its precondition (more formally, the app_A formula of Chapter 2) takes the value *false* (i.e., the assertion is never applicable) or if its postcondition (more formally, the sat_A formula of Chapter 2) takes the value *true*. The postcondition of the first assertion of our example takes the value *true*. Thus, there is no need to consider such an assertion in run-time checking, and we can rewrite it with the tag `checked`. An assertion is violated if its precondition takes the value *true* and its postcondition takes the value *false*.[13] This happens to the second assertion in our example. Thus, we can rewrite it with the tag `false`. Whenever an assertion is detected to be false at compile-time, in addition to being rewritten with the `false` tag, the preprocessor also issues an error message. This allows the user to be aware of an incorrectness problem without looking at the assertions obtained by compile-time checking.

If it is not possible to modify the tag of an assertion, then such assertion is left as a `check` assertion, for which run-time checks might be generated. However, as the assertion may have been simplified, this allows reducing the number of properties which have to be checked at run-time.

Example 3.3.5. The final result of compile-time checking of assertions is:

```
:- checked success p(X,Y) =>  (ground(X),ground(Y)).
:- false   success p(X,Y) =>  (list(X,int), list(Y,int)).
:- check   comp    p(X,Y) +   terminates.
```

[13] There is a caveat in this case due to the use of over-approximations in program analysis. It may be the case that a compile-time error is issued which does not occur at run-time for any valid input data. This is because though any activation of the predicate would be erroneous, it may also be the case that the predicate is never reached in any valid execution but analysis is not able to notice this. However, we believe that such situations do not happen so often and also the error flagged is actually an error of the program code. Though it can never show up in the current program it could do so if the erroneous part of the program is used in another context (in which it is actually used) in another program.

where the third assertion still has the tag `check` since it is not guaranteed to hold nor to be violated. Note also that the two assertions whose tag has changed appear as in the original version rather than the simplified one. The preprocessor does so as we believe it is more informative.

One important consideration about compile-time checking is that, for a given analysis, not all properties are equally simple to reduce at compile-time to either *true* or $false$. Unless otherwise stated, we assume that, as is usually the case, analysis computes over-approximations (but it is straightforward to develop dual solutions for under-approximations). We first study the case of reducing a property to *true* and then the case for reducing it to $false$.

A declaration of the form '`:- InfSys prop Prop/n.`' implicitly states that there is some abstract description λ_{TS}(`Prop/n`) computable by the static analysis `InfSys` such that for all stores θ in $\gamma(\lambda_{TS}($`Prop/n`$))$ the property `Prop/n` holds. In such case, if at the corresponding context, analysis computes a description λ such that $\gamma(\lambda) \subseteq \gamma(\lambda_{TS}($`Prop/n`$))$ then the property is clearly guaranteed to hold for any store in $\gamma(\lambda)$. This process corresponds to abstractly executing the property to the value *true* [3.41, 3.40].

Another way of proving that a property holds may be provided by the existence of a declaration of the form '`:- proves Prop(X) : Prop2(X).`' (see Chapter 2), which indicates that in order to prove `Prop(X)` it is sufficient to prove `Prop2(X)`. Note that these `proves` declarations may be chained to any length. If we reach some `PropN(X)` for which a declaration '`:- InfSys prop PropN.`' exists, then we can try to prove it as described above.

Regarding reducing a property to $false$, analyses based on over-approximations are not so appropriate. Assume that the result of analysis is a description λ which is an over-approximation of the set of stores which may occur during the execution of a given program construct. The fact that there are elements in $\gamma(\lambda)$ in which a property AF does not hold does not guarantee that AF actually does not hold during execution. It is possible that such elements correspond to stores approximated by $\gamma(\lambda)$ but which do not actually occur during execution of the given program construct. They might have been introduced due to the loss of precision of the analysis. In any case, if the property AF does not hold in *any* of the elements of $\gamma(\lambda)$ then it does not hold in any of the actual stores either. This is a sufficient condition for the property to be false.

Despite the discussion above, and as we will see in the coming example, it is important to mention that it is possible to reduce a property `Prop/n` to $false$ using an inference system `InfSys` without the need of any assertion of the form '`:- InfSys prop Prop/n`' which declares `Prop/n` as *directly provable* in `InfSys`, and even if no `disproves` assertion exists for `Prop/n`. This is very convenient, since in order to add an assertion '`:- InfSys prop Prop/n`' we need to know a suitable λ_{TS}(`Prop/n`) and this requires a good understanding of both the inference system `InfSys` and the property under consideration `Prop/n`. As a result of being able to reduce very general properties to $false$, the preprocessor can detect incorrectness problems at compile-time even if the properties used in the assertions have not been written with any static inference system in mind.

Example 3.3.6. Consider a property `sorted_num_list` to check for sorted lists of numbers (defined as in Section 3.6), and mode and regular type analyses. Although the property cannot be proved with only (over approximations) of mode and regular type information, it may be possible to prove that it does *not* hold (an example of how properties which are not provable of any analysis can also be useful for detecting bugs at compile-time). While the regular type analysis cannot capture perfectly the property `sorted_num_list(T)`, it can still approximate it (by analyzing the definition) as `list(T,num)`. If type analysis for the program generates a type for `T` which is incompatible with `list(T,num)`, then a definite error symptom is detected.

Similarly to the case of reducing properties to *true*, a declaration of the form '`:- disproves Prop(X) : Prop2(X).`' can be used to disprove `Prop(X)` if we can prove `Prop2(X)`. Again, we can also have a chain of `proves` declarations which may allow proving `Prop2(X)` and thus disproving `Prop(X)`. Further discussion on how to deal with the different kinds of properties and the impact of approximations can be found in [3.7, 3.39].

Properties that are directly provable in an inference system `InfSys` are the main targets for abstract execution when using `InfSys` as static inference system, since it is often possible to either prove them or disprove them, provided that `InfSys` is accurate enough. This is the case for the properties `ground`, `list(int)` and `does_not_fail` in the example above. Note that, if the analysis `InfSys` is *precise* (in the sense that the abstract operations do not lose information beyond the abstraction implied by the abstraction function used [3.17]) and, obviously, terminates, then there are properties which can be reduced to either *true* or $false$ in all cases, i.e., for which the analysis is a complete inference system. In fact, in systems conceived for compile-time checking only, the properties admitted in assertions are usually restricted to those which are (almost) *perfect observables* of the analysis available, i.e., they can (almost) always be proven or disproven. This is not our case, and nevertheless, we have seen how interesting properties –which are not perfect observables– can still be used to detect incorrectness symptoms.

Though the process just shown simplifies predicate assertions using the information static analysis has inferred for the global (call and) success states for the predicate, in order to perform compile-time checking of program-point assertions the inference system needs to be able to compute information at each program point. In fact, both `shfr` and `regtypes` do so. It is also important to mention that it is also very useful to perform compile-time checking of predicate assertions at each program point. The interest in doing this is because many different calls to a predicate may exist in the program and it is likely that only some of them are incorrect. Thus, as some analyses summarize all possible calls in one description (by losing precision), it is not possible with these analyses to detect that an assertion is violated in *some* calls to the predicate. One way to solve this problem is to compute analysis information at each program point and then compare the analysis information at each program point with the assertions for the corresponding predicate called at that particular program point.[14]

Be it at the predicate level or at the program-point level, the fact that an assertion remains in status `check` cannot be taken as an indicator of an incorrectness problem, as is the case in strong type systems, since it may be due to the lack of an static analysis for which the given property is provable, or to the loss of precision of the analyzer. In any case, issuing a warning might be helpful, but (automatic) checking would have to be performed at run-time.

3.4 Run-Time Checking

The aim of run-time checking of assertions is to detect those assertions which are violated during the execution of the program for the data being explored. Such checking involves:

1. At each execution state it must be determined which are the assertions (if any) which *affect* the corresponding state together with the logic formulae which have to be evaluated at the correspondig store.
2. Actually evaluating the required atomic formulae AF, i.e. computing $eval(AF, \theta, P, IS)$.
3. Obtaining the truth value of the logic formulae as a whole.
4. Obtaining the truth value of the assertion as a whole. If the assertion is detected to be violated error messages should be issued.

As discussed earlier (Figure 3.2), run-time checking in the preprocessor is implemented as a program transformation which adds new code to the program as needed to perform the required checking during execution of the program. This program transformation, presented in Section 3.4.2 below, is in charge of solving item 1 above. As discussed in Chapter 2, item 2 above, i.e., evaluation of the atomic formulae, must be performed by a suitable inference system IS. We discuss how this evaluation is performed in our preprocessor in Section 3.4.1, under the assumption that we use the underlying CLP system as the inference system IS. This section also discusses how to obtain the truth value of the logic formulae as a whole, i.e., item 3 above. Finally, in Section 3.4.2 we present the overall program transformation and a series of predicates which are in charge of obtaining the truth value of each assertion as a whole and of issuing errors when an assertion is detected to be violated.

[14] Furthermore, since the analysis is multi-variant, a single program point may be reflected in analysis by several descriptions. Thus, it is in principle possible to perform assertion checking and diagnosis at a finer-grained level.

The code of such predicates is independent of the particular program being checked and is stored in a library program named `rtchecks` which is loaded by the the transformed program (which requires such library in order to run correctly). If the checking proves that an assertion is violated, a *concrete*[15] incorrectness symptom is detected and some kind of error message is given to the user. A procedure for localizing the cause of the error, such as standard or declarative diagnosis can then be started. It is out of the scope of this chapter to discuss how program diagnosis should be performed. However, techniques such as declarative debugging [3.43, 3.3, 3.20, 3.21] (see Chapter 5) or more traditional interactive debuggers [3.10, 3.22] may be applied. Correctness of the transformation requires that the transformed program only produce an error message if the specification is in fact violated.

3.4.1 Evaluating Atomic Logic Formulae

As discussed in Chapter 2, one of the design decisions of our assertion language is that property predicates are defined in the source language. Given this, and although the assertion language design leaves the choice of implementation for the inference system open, it seems interesting to study the viability of using the underlying (C)LP system to perform the run-time checks, i.e., to study whether $eval(AF, \theta, P, IS)$ can be reduced to true or false with IS being the (C)LP system in which the program is running.

This introduces some simplifications in the problem: since IS is given, θ is the current store, and P is the program itself, then we can simplify $eval(AF, \theta, P, IS)$ to simply $eval(AF)$. Since AF is a predicate of the program, then the answer to this question could be obtained, at least at a first level of approximation, simply by calling AF directly and seeing whether it succeeds or fails. This raises at least two issues. First, the fact that the system will interleave calls to the code which defines each property predicate with the execution of the program imposes in practice some limitations on such code. Essentially, we would like the behaviour of the program not to change in a fundamental way independently of whether the program contains run-time checks or not. The second issue is whether we can use the result of the execution of AF in the current store as the value of $eval(AF)$.

Regarding the first issue above, while we can tolerate a degradation in execution speed, turning on run-time checking should not introduce non-termination in a program which otherwise terminates. Also, checking a property should not change the answers computed by the program or produce unexpected side-effects (such as new program errors, other than those from assertion violations). In light of this, a reasonable set of requirements on the definitions of the predicate properties is the following:

[15] As opposed to *abstract* incorrectness symptoms, which are the ones detected by compile-time checking.

1. The execution of the code which defines the property predicate should terminate, and it should do so with success or failure, not error, *for any possible input.*
2. The code defining a property should not perform input/output, add or delete clauses, etc., which may interfere with the program behaviour.
3. The execution of a property should not further instantiate its arguments or add new constraints, it may not change the store θ seen by the subsequent part of the execution.

In practice, these conditions can be relaxed somewhat in order to simplify the task of writing property predicates. In particular, and as we will see, the program transformation that we present below guarantees requirement 3, i.e., run-time checking is guranteed not to introduce any undesired constraints, independently of how the property predicates are written. Thus, condition 3 can be ignored in practice. Also, some basic checks on the code defining property predicates are enough in most cases for requirement 2, i.e., the system can easily be made to reject a property predicate which does not satisfy condition 2. However, the user is responsible for guaranteeing termination of the execution of the properties defined. To this end, our system assumes that property predicates declared with `prop` (i.e., those of the execution states) will always terminate for any possible calling state.

We now turn to the second issue above, i.e., whether the result of executing AF in the current store can be used as the value of $eval(AF)$. Recall that the assertion language (Chapter 2) specifies that the checking of properties about execution states has to guarantee that they are treated as instantiation checks, unless they are qualified by `compat`, in which case they should be treated as compatibility checks. The fact that the assertion language allows writing the definitions of properties in such a way that they can be used both in instantation and compatibility checks, imposes the need for some machinery beyond simply calling AF. It turns out that performing instantiation checks corresponds to performing what is referred in (C)LP as *entailment* and *disentailment* tests [3.32, 3.39]. A constraint is *entailed* by a constraint store if the constraint is implied by the store. Conversely, a constraint is *disentailed* by a constraint store if it is inconsistent with the store. We extend the notion of entailment and disentailment, originally defined on constraints, to property predicates. We say that a property predicate is entailed by a constraint store if its execution succeeds without affecting the store, i.e., any constraint added to the store only affects variables which are local to the definition of the predicate. Also, we say that a property predicate is disentailed by a constraint store if its execution fails. An instantiation property is true if it is entailed, and false if it is disentailed.

Given these definitions, the question then is how can entailment and disentailment tests be performed by evaluation of the property formulae and how the characteristics of the CLP system might affect this evaluation. Figure 3.5 lists definitions for `entailed/1` or `disentailed/1` which achieve this task,

```
disentailed((LogForm,LogForms)):- !,
        ( disentailed(LogForm); disentailed(LogForms) ).
disentailed((LogForm;LogForms)):- !,
        disentailed(LogForm), disentailed(LogForms).
disentailed(compat(LogForm)):- !,
        system_dependent_incompatible(LogForm).
disentailed(AtomForm):-
        \+ \+ system_dependent_disentailed(AtomForm).
disentailed(AtomForm):-
        disproves(AtomForm,SufficientCond),
        entailed(SufficientCond).

entailed((LogForm,LogForms)):- !,
        entailed(LogForm), entailed(LogForms).
entailed((LogForm;LogForms)):- !,
        ( entailed(LogForm); entailed(LogForms) ).
entailed(compat(LogForm)):- !,
        system_dependent_compatible(LogForm).
entailed(AtomForm):-
        \+ \+ system_dependent_entailed(AtomForm).
entailed(AtomForm):-
        proves(AtomForm,SufficientCond),
        entailed(SufficientCond).
```

Fig. 3.5. Entailment and Disentailment of Logic Formulae

i.e., properties are treated as instantiation checks unless they are marked as `compat`, in which case they are treated as compatibility checks, and also make sure that checking does not affect program execution by posing any additional constraints on the store.

The implementation of Figure 3.5 also takes care of an additional issue: guaranteeing that the possible incompleteness of the constraint solver does not affect correctness of the dynamic assertion checking. By incompleteness we mean the fact that the solver may fail to detect that some state (constraint store) is inconsistent when it is indeed inconsistent. This may happen because of constraints which the solver decides to delay, as they are hard to treat in the current state, and will be taken into account when more information is available. Note that a simple sufficient condition for a property to be disentailed is that the execution of the predicate defining it actually fails, which means that the constraints it imposes are inconsistent with the store. If the solver is incomplete, properties which are false may go undetected, and preconditions which in fact do not hold might be deemed to hold, which is obviously incorrect.

Moreover, finding a sufficient condition for entailment of a property is not as easy as with disentailment. In fact, many CLP systems do not even have an entailment test for constraints. Also, checking that the constraint store remains the same after the execution of a property predicate (i.e., that its execution has not added constraints to the store) may not be an easy task, as store comparison is an operation which is usually not available. Therefore, these checks have to be defined separately for each CLP system. I.e., system implementors should provide suitable definitions for the predicates `system_dependent_entailed`, `system_dependent_disentailed`, `system_dependent_compatible`, and `system_dependent_incompatible`. Possible implementations for a particular constraint system are given in Section 3.5.2.

Also, note that there are properties for which no *accurate* definition can be written in the underlying language, and therefore it is difficult that executable definitions are given for them. For these properties, an *approximate* definition may be given, and this approximation should be correct in the usual sense that all errors flagged should be errors, but there may be errors that go undetected. This can be done in the assertion language with `proves` and `disproves` assertions. These assertions are translated during the transformation of the program into two tables of facts, `proves/2` and `disproves/2`, which are then considered during run-time checking by the appropriate clauses of the predicates `entailed/1` and `disentailed/1`, as seen in the code presented above.

Finally, the situation is slightly different for properties of the computation, i.e., those property predicates declared with `cprop`. These predicates may have to reconstruct the computations of which they have to decide if the property they define holds or not. Since the necessary part of the computation required may be an infinite object, it is possible that the process of reconstructing such computation does not terminate, in which case the execution of the property predicate will not terminate either. Thus, we admit that the predicates for properties of the computation do not terminate, *provided* that the execution of the corresponding computation does not terminate either. Note that in this case run-time checking will not introduce non-termination into the program, in the sense that if the execution of the original program terminates then its execution with run-time checking will also terminate. It is also worth mentioning that the run-time translation presented is valid provided that the computation on which a property has to be checked does not perform side-effects (or the property is written in such a way that it captures calls to side-effects and avoids them). Otherwise, such side-effects may be performed more than once since after checking the assertion the computation is performed anyway.

3.4.2 A Program Transformation for Assertion Checking

This behaviour can be acomplished by a program transformation which, basically, precedes each call to a predicate involved in a predicate assertion with calls to the atomic properties in the precondition of the assertion, and postpones calls to the atomic properties in the postcondition until after success of the call.

We now provide a possible scheme for translation of a program with assertions into code which will perform run-time checking. Our aim herein is not to provide the best possible transformation (nor the best definition of auxiliary predicates used by it), but rather to present simple examples with the objective of showing the feasibility of the implementation and hopefully clarifying the approach further.

Given a predicate `p/n` for which assertions have been given, the idea is to replace the definition of `p/n` so that whenever `p/n` is executed the assertions for it are checked and the actual computation originally performed by `p/n` is now performed by the new predicate `p_new/n` (where `p_new` stands for a fresh atom which is guaranteed not to concide with other predicate names). I.e., given the definition of a predicate `p/n` as:

```
p(t11,...,t1n):- body_1.
          ...
p(tm1,...,tmn):- body_m.
```

it gets translated into:

```
p(X1,...,Xn):-
        check_assertions_and_execute ``p_new(X1,...,Xn)''.

p_new(t11,...,t1n):- body_1.
              ...
p_new(tm1,...,tmn):- body_m.
```

I.e., the definition of `p_new/n` corresponds to the definition of `p/n` in the original program and is independent of the assertions given for `p/n`. The checks present in the new definition of predicate `p/n` depend on the existing assertions for such predicate.

Success Assertions. A possible translation scheme for `success` assertions with a precondition is the following. Let $A($`p/n`$)$ represent the set of assertions for predicate `p` of arity `n`. Let `RS` be the set $\{(p(Y_1, ..., Y_n), Precond, Postcond)$ s.t. '`:- success p(Y1,...,Yn) :` *Precond* `=>` *Postcond*' $\in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):-
        prec(RS,p(X1,...,Xn),S),
        p_new(X1,...,Xn),
        postc(S,p(X1,...,Xn)).
```

where `prec/3` collects the elements in `RS` s.t. *Precond* definitely holds at the calling state to `p/n`. A possible implementation of the `prec/3` predicate is the following:

```
prec([],_Goal,[]).
prec([(Pred_desc, Precond, Postcond)|RS],Goal,NRS):-
        test_entailed(Precond,Pred_desc,Goal),!,
        NRS = [(Pred_desc, Precond, Postcond)|MoreRS],
```

```
        prec(RS,Goal,MoreRS).
prec([_|RS],Goal,NRS):-
        prec(RS,Goal,NRS).

test_entailed(LogForm,Pred_desc,Goal):-
        \+(\+((Pred_desc=Goal, entailed(LogForm)))).
```

note that those assertions whose precondition cannot be guaranteed to hold are directly discarded.

Finally, `postc/2` checks whether the *Postcond* of each assertion collected by `prec/3` holds or not. If it does not hold then the corresponding assertion is definitely violated and usually an error message will be given to the user (actually, in CiaoPP an exception is raised), and optionally, computation halted. If either they hold or they cannot be guaranteed not to hold computation will generally continue as usual. A possible implementation follows:[16]

```
postc([],_).
postc([(Pred_desc,Precond , Postcond)|_Cs],Goal):-
        Pred_desc = Goal,
        disentailed(Postcond),
        write('ERROR: for Goal '),
        write(Pred_desc), nl,
        write('with Precondition '),
        write(Precond), nl,
        write(' holds but Postcondition '),
        write(Postcond),
        write(' does not.'),nl,
        fail.
postc([_|Cs],Goal):-
        postc(Cs,Goal).
```

note that the call `p(X1,...,Xn)` is passed as an argument to both `prec/3` and `postc/2`. However, such call is not executed but rather it is used to pass the values of the arguments `X1,...,Xn` just before and after executing `p_new(X1,...,Xn)`. Something similar happens in the `calls/2` predicate introduced below.

Calls Assertions. A possible translation scheme for `calls` assertions is the following. As before, let A(`p/n`) be the set of assertions for predicate `p` of arity `n`. Let `C` be the set $\{(p(Y_1,...,Y_n), Precond)$ s.t. '`:- calls p(Y1,...,Yn) :` *Precond*' $\in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):-
        calls(C,p(X1,...,Xn)),
        p_new(X1,...,Xn).
```

where `calls/2` checks whether the preconditions for the predicate hold or not. If it detects that some *Precond* in `C` does not hold for some call to the

[16] In the code we use for simplicity calls to `write/1`. However in the actual implementation this is better implemented by raising exceptions (using `throw`).

predicate then the corresponding assertion is violated. A possible implementation of the calls/2 predicate follows:

```
calls([],_Goal).
calls([(Pred_desc, Precond)|_Calls],Goal):-
        Pred_desc = Goal,
        disentailed(Precond),
        write('ERROR: undefined call for predicate '),
        write(Goal), nl,
        fail.
calls([_|Calls],Goal):-
        calls(Calls,Goal).
```

Comp Assertions. A possible translation scheme for comp assertions is the following. Let RC be the set $\{(p(Y_1,...,Y_n), Precond, Comp\_prop)$ s.t. ':- comp p(Y1,...,Yn) : *Precond* + *Comp_prop*' $\in A(p/n)\}$. Then the translation is:

```
p(X1,...,Xn):-
        check_comp(RC1,p(X1,...,Xn)),
        p_new(X1,...,Xn).
```

where in RC1 we have added to each property of the computation in RC the goal p_new(X1,...,Xn) as the first argument.

Example 3.4.1. Given RC= {(p(Y1,...,Yn), ground(Y1),deterministic)} then RC1= [(p(Y1,...,Yn), ground(Y1), deterministic(p_new(X1,...,Xn)))].

The check_comp predicate aims at checking *Comp_prop* for those comp assertions whose precondition holds. If *Comp_prop* is guaranteed not to hold then the corresponding assertion is definitely violated. A possible implementation follows:

```
check_comp([],_Goal).
check_comp([(Pred_desc,  Precond, CompProp)|_RC],Goal):-
        test_entailed(Precond,Pred_desc,Goal),
        Pred_desc = Goal,
        system_dependent_incompatible(CompProp),
        write('ERROR: for Goal '),
        write(Pred_desc), nl,
        write('with Precondition '),
        write(Precond), nl,
        write(' holds but CompProp '),
        remove_int_goal(CompProp,NCompProp),
        write(NCompProp),
        write(' does not.'), nl,
        fail.
check_comp([_|RC],Goal):-
        check_comp(RC,Goal).
```

Note that in this case in order to guarantee that *Comp_Prop* actually does not hold, system_dependent_incompatible rather than disentailed is used. This is because in contrast to properties of execution states, properties of the computation are not treated by default as instantiation properties since the computation itself may add constraints and the property can still be considered to hold. Such constraints will be eliminated from the store by forcing backtracking (see the definition of `system_dependent_incompatible` in Section 3.5.2).

Program-Point Assertions. Since they are part of the program, in order to execute a program which contains program-point assertions, a definition of the (meta) predicate `check/1` must be provided. A possible definition is the following:

```
check(LogForm):-
        disentailed(LogForm), !,
        write('ERROR: false program point assertion'),
        nl, write(LogForm),nl.
check(_).
```

where the definition of `disentailed` is given below. If run-time checking is not desired there are two possibilities, either the preprocessor can simply remove the program-point assertions from the program or, as already mentioned in Chapter 2, the alternative definition for `check/1` is used:

```
check(_).
```

3.5 Customizing the Preprocessor for a CLP System: The CiaoPP and CHIPRE Tools

We have implemented the schema of Figure 3.3 as a *generic preprocessor*. This genericity means that different instances of the preprocessor for different CLP systems can be generated in a straightforward way. One reason for this is that within the DiSCiPl Project, several platforms exist (Ciao, CHIP, PrologIV, etc.) for which the preprocessor is of use. The preprocessor is parametric w.r.t. the set of system predicates, small syntax differences (mainly definition of operators) and the set of analyzers available. The generic preprocessor is a stand-alone application written in Ciao Prolog, which is a public domain next generation logic programming system developed by the Clip group at UPM.

Currently, two different experimental debugging environments have been developed using this generic preprocessor:

- CiaoPP [3.8, 3.30], the Ciao system preprocessor, developed by the Clip group at UPM, and
- `CHIPRE` [3.9], a preprocessor for CHIP developed also by the Clip group at UPM, in collaboration with Pawel Pietrzak at Linköping University (adaptation of Gallagher's type analysis to the CLP(FD)language used by CHIP), and Helmut Simonis at COSYTEC (initial customization assertions for supporting the CHIP builtin predicates –see Section 3.5.1– and development of a graphical user interface to the tool).

In order to port the preprocessor to another CLP system we basically require the following:

– The preprocessor should understand the syntactic and semantic particularities of the CLP system. Understanding the syntactic particularities generally amounts to the definition of a set of operators which are predefined in the language. Understanding the semantic particularities amounts to providing a description of the system predicates in the given CLP. This issue is further discussed in Section 3.5.1 below. This allows both performing accurate static analysis and checking calls to system predicates at compile-time.
– The CLP system should be able to read programs with assertions. This is needed since in order to exploit the full power of the preprocessor users should add assertions to their programs. This can be solved by adding to the CLP system a library program which contains the definitions of operators required for reading assertions. We provide such a library, called `assertions`, together with the distribution of the preprocessor.
– The CLP system should be able to run programs which have been subject to the run-time checking translation scheme. Such transformation includes in the program calls to predicates whose definition is also provided in a library program called `rt_checks`. Most of the code of this program is independent of the CLP system being used. However, depending on the system, some predicates have to be customized for the particular system being used. Much in the same way as assertions for builtin predicates, the definitions of such predicates are supposed to be provided by system developers rather than by users of the preprocessor. More details on this are presented in Section 3.5.2 below.

It is also important to mention that another way of customizing the preprocessor is by integrating new analyses in addition to the existing ones. This can be done by defining new domains for analysis for the generic analysis engine i.e., PLAI [3.38], in the preprocessor. Due to space limitations, we do not go into the details on how to define new analysis domains here.

3.5.1 Describing Builtins using Assertions

Assertions for builtin predicates should be provided by the system programmers to allow compile-time checking of the system predicates in the preprocessor (and also facilitate their run-time checking, as discussed before). The system predicate assertions must be available at the time of installing the preprocessor, so that it is configured for the particular language, and therefore, it understands the builtin predicates of the language. It is convenient to provide assertions that describe:

– Their success states. This is useful for improving the information obtained by static analysis. Note that many of the builtin predicates may be written

in other languages, such as C. Therefore, it is not easy to automatically analyze their code with the analyzers of the preprocessor, which are designed for CLP languages.
- Their required calling states. This is useful for performing compile-time checking of the calls to the system predicates. Unless this is done, checking of such calls has to be done at run-time.

The former can be described using `trust success` assertions, the latter with `check calls` assertions. They should define all and the only possible uses of a predicate.[17] If the preprocessor can determine that a system predicate is used in a way which is different from those described by the assertions for it, its behaviour is deemed to be unpredictable and a compile-time error is issued.

Example 3.5.1. Consider the `cumulative/8` builtin global constraint of CHIP. A call of the form:

```
cumulative(Starts,Durations,Resources,Ends,Surfaces,High,End,Int)
```

states, in its most simple form, that the cumulative use of a resource by all tasks with starts dates (in the list of finite domain ranges of) `Starts`, durations (in the list of ranges of) `Durations`, and resource use (in the list of ranges of) `Resources` is below the (range of) limit `High`. In this use of the constraint the arguments `Ends`, `Surfaces`, `End`, and `Int` are assigned the value `unused`. However, in its most general form, the cumulative constraint can be used with any of `Ends` and `End` assigned a list of ranges, `Int` a range, and/or `Surfaces` a list of integers.

To describe the behaviour of the cumulative constraint in terms of, e.g., regular types, one needs to use the type "list of finite domain variables" (i.e., `list(anyfd)`), the type "assigned value `unused`", as well as the types which describe the rest of the arguments. These types may be defined with the regular predicates:

```
:- regtype prop notused/1.
notused(unused).

:- regtype prop fd_or_unused/1.
fd_or_unused(unused).
fd_or_unused(X) :- anyfd(X).

:- regtype prop fdlist_or_unused/1.
fdlist_or_unused(unused).
fdlist_or_unused(X) :- list(X,anyfd).

:- regtype prop intlist_or_unused/1.
intlist_or_unused(unused).
intlist_or_unused(X) :- list(X,int).
```

[17] For this purpose `pred` assertions can be used, which are translated into one `check calls` and several `trust success` assertions.

so that the following assertion describes all the possible uses of the cumulative constraint:

```
:- trust pred cumulative/8
         : list(anyfd) * list(anyfd) * list(anyfd)
         * fdlist_or_unused * intlist_or_unused * anyfd
         * fd_or_unused * fdlist_or_unused
        => list(anyfd) * list(anyfd) * list(anyfd)
         * fdlist_or_unused * intlist_or_unused * anyfd
         * fd_or_unused * fdlist_or_unused .
```

However, it is also convenient to provide descriptions for particular uses of the predicates, since this may allow the analyzers to improve accuracy. Detailed descriptions of different uses provide more information than a unique global description of all possible uses. In order to do this, one could have added success assertions for some (or all) of the possible uses of the predicate.

Example 3.5.2. The abovementioned use of the cumulative constraint (which corresponds to a typical use in manpower resource restricted scheduling), in which the arguments named above `Ends`, `End`, `Int`, and `Surfaces` are not used, is described by the (additional) assertion:

```
:- trust success cumulative/8
         : list(anyfd) * list(anyfd) * list(anyfd)
         * notused * notused * anyfd * notused * notused.
        => list(anyfd) * list(anyfd) * list(anyfd)
         * notused * notused * anyfd * notused * notused .
```

Though it may be argued that writing the assertions describing the system predicates is a tedious task, such task is not supposed to be performed by the users of the preprocessor, which can assume that such assertions are already present in the preprocessor. Also, this task is in any case definitely much simpler than writing an analyzer from scratch or adapting an existing one written for another CLP system.

However, a disadvantage of describing system predicates in terms of assertions rather than in some lower-level description is a relative loss of efficiency. This is because, as is usual in software, the more general a piece of software is, the less efficient. In fact, software can often be optimized by *specializing* [3.35, 3.25] it w.r.t. some particular case. In our case, we would like our system to be at the same time easily portable to different systems and as efficient as if it had been designed with a particular system in mind. With this aim, we have implemented a program called `builtintables` which takes as input the assertions describing the system predicates and converts such information into the internal format of the analyzers. This is conceptually equivalent to specializing an analyzer to a particular set of system predicates. However, rather than using a general purpose specializer, we follow the simpler to implement *generating extension approach* of partial evaluation in that the above mentioned program only knows how to specialize the analyzer w.r.t. the assertions describing the system predicates. The `builtintables` program is only executed whenever the assertions for system predicates are modified.

3.5.2 System Dependent Code for Run-Time Checking

In order to perform run-time checking, a library `rtchecks` should be provided by the system. Most of the predicates in this library are independent of the underlying CLP system used, and possible implementations of such predicates have been given above. However, there are four predicates whose definition was not provided. This is because they depend on the particular CLP system being used.

The role of the system dependent predicates is basically to determine whether properties hold (are entailed) or not (are disentailed). Since individual properties may appear with the `compat` qualifier, we also need predicates for testing for compatibility and incompatibility.

We have chosen to define the system dependent predicates for the case of using Prolog as underlying system. This is both because we assume most readers are familiar with Prolog and because of the "good behaviour" of the Herbrand solver. However, we indicate which parts of the definitions are valid in any system and which other ones can only be used under certain assumptions.

The predicate `system_dependent_entailed` should succeed only if we can guarantee that `Prop` is implied by the store. This is probably the most system-dependent predicate of the four. Some constraint systems may have a complete entailment test, while others may not. In Herbrand this can be done by checking that `NProp`, which is a copy of `Prop`, succeeds and no new constraints (bindings) have been generated during its execution. This is checked by predicate `instance` which succeeds iff `Prop` is an instance of `NProp`:

```
system_dependent_entailed(Prop):-
        copy_term(Prop,NProp),
        call(NProp), !, instance(Prop,NProp).
```

Next we provide a definition for predicate `system_dependent_compatible`, which is valid for any constraint system which is "quick rejecting" (also called immediate), as is the case in Herbrand. In those systems, whenever we add a constraint which is inconsistent with the current store, this is immediately detected and a failure is issued. Note that this is not always the case since the system may delay the consistency check until further constraints added to the store allow performing the check in a simple way:

```
system_dependent_compatible(Prop):-
        \+(\+(call(Prop))).
```

Next we provide a definition of predicate `system_dependent_disentailed` which in contrast to the other three predicates is defined by two clauses. Each one provides a sufficient condition for disentailment. The first one corresponds to the case of incompatibility (whose definition appears later). Clearly, if `Prop` is incompatible with the current store it is also disentailed (but not the other way around). The second clause corresponds to the case in which execution of the property succeeds but it explicitly requires additional information in order to succeed. An example of this is the property `list(X)` which is an instantiation type which requires `X` to be instantiated (at least) to a list skeleton, and with the store containing the information `X = [1|Y]`. In this case, `X` is compatible with a list (and thus the first clause would fail) but it is not actually a list. A call to `list([1|Y])` would succeed but would for example add the extra information that `Y=[]`:

```
system_dependent_disentailed(Prop):-
        system_dependent_incompatible(Prop),!.
system_dependent_disentailed(Prop):-
        copy_term(Prop,NProp),
        call(NProp), !, \+(instance(Prop,NProp)),
        instance(NProp,Prop).
```

Finally, for predicate `system_dependent_incompatible` we provide a definition which is valid in any system:

```
system_dependent_incompatible(Prop):-
        \+(call(Prop)).
```

3.6 A Sample Debugging Session with the Ciao System

In order to illustrate from the users point of view several of the points made in the previous sections we now present a sample debugging session with a concrete tool, CiaoPP, the Ciao system preprocessor [3.30, 3.8] which is currently part of the programming environment of Ciao, and which, as mentioned before, is an instance of the generic preprocessor presented in this chapter. CiaoPP uses as analyzers both the LP and CLP versions of the PLAI abstract interpreter [3.38, 3.6, 3.27] and adaptations of Gallagher's regular type analysis [3.26], and works on a large number of abstract domains including moded types, definiteness, freeness, and grounding dependencies (as well as more complex properties, such as bounds on cost, determinacy, non-failure, etc., for Prolog programs).

Basic Static Debugging. The idea of using analysis information for debugging comes naturally after observing analysis outputs for erroneous programs. Consider the program in Figure 3.6. The result of regular type analysis for this program includes the following code:

```
:- true pred qsort(A,B)
         : ( term(A), term(B) )
        => ( list(A,t113), list(B,t118) ).

:- regtype prop t113/1.
```

```
:- module(qsort, [qsort/2], [assertions]).

qsort([X|L],R) :-
        partition(L,L1,X,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R2,[x|R1],R).
qsort([],[]).

partition([],_B,[],[]).
partition([e|R],C,[E|Left1],Right):-
        E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E >= C,   partition(R,C,Left,Right1).

append([],X,X).
append([H|X],Y,[H|Z]):- append(X,Y,Z).
```

Fig. 3.6. A tentative qsort program

```
t113(A) :- arithexpression(A).
t113([]).
t113([A|B]) :- arithexpression(A), list(B,t113).
t113(e).

:- regtype prop t118/1.
t118(x).
```

where `arithexpression` is a library property which describes arithmetic expressions. Two new names (`t113` and `t118`) are given to types inferred, and their definition included, because no definition of such types were found visible to the module. In any case, the information inferred does not seem compatible with a correct definition of `qsort`, which clearly points to a bug in the program.

In order to debug the program, we add to it a declaration of its valid queries as follows:

```
:- entry qsort(A,B) : (list(A, num), var(B)).
```

Turning on compile-time error checking and selecting the `regtype` and `shfr` static analyses we obtain the following messages:

```
WARNING: Literal partition(L,L1,X,L2) at qsort/2/1/1
                 does not succeed!
ERROR: Predicate E>=C at partition/4/3/1 is not called as expected:
         Called:   num>=var
         Expected: arithexpression>=arithexpression
```

The first message warns that all calls to `partition` will fail, something normally not intended (e.g., in our case). The second message indicates a wrong call to a builtin predicate, which is an obvious error. This error has been detected by comparing the information obtained by the `shfr` inference system, which at the corresponding program point indicates that `E` is a free variable, with the assertion:

```
:- check calls A<B : (arithexpression(A), arithexpression(B)).
```

which is present in the default builtins module, and which implies that the two arguments to `</2` should be bound to arithmetic expressions, and thus ground. The message signals an *abstract* incorrectness symptom, indicating that the program does not satisfy the specification given (that of the builtin predicates, in this case). Checking the indicated call to `partition` and inspecting its arguments we detect that in the definition of `qsort`, `partition` is called with the second and third arguments in reversed order – the correct call is `partition(L,X,L1,L2)`.

After correcting this bug, we proceed to perform another round of compile-time checking, which produces the following message:

```
WARNING: Clause 'partition/4/2' is incompatible with its call type
         Head:      partition([e|R],C,[E|Left1],Right)
         Call Type: partition(list(num),num,var,var)
```

This time the error is in the second clause of `partition`. Checking this clause we see that in the first argument of the head there is an `e` which should be `E` instead. Compile-time checking of the program with this bug corrected does not produce any further warning or error messages.

Validation of User Assertions. In order to be more confident about our program, we add to it a partial specification of the program in the form of the following `check` assertions:

```
:- shfr prop ground/1.
:- regtype prop list/2.
:- prop sorted_num_list/1.
sorted_num_list([]).
sorted_num_list([X]):- number(X).
sorted_num_list([X,Y|Z]):-
        number(X), number(Y), X<Y, sorted_num_list([Y|Z]).

:- calls qsort(A,B) : list(A, num).                         % A1
:- success qsort(A,B)  => (ground(B), sorted_num_list(B)).  % A2
:- calls partition(A,B,C,D) : (ground(A), ground(B)).       % A3
:- success partition(A,B,C,D) => (list(C, num),ground(D)).  % A4
:- calls append(A,B,C) : (list(A,num),list(B,num)).         % A5
```

where we also use a new property, `sorted_num_list`, defined in the module itself. We then ask CiaoPP to check them, by comparing them with the information inferred by analysis, which produces:

```
:- checked calls qsort(A,B) : list(A,num).                          %A1
:- check success qsort(A,B)  => sorted_num_list(B).                 %A2
:- checked calls partition(A,B,C,D) : (ground(A),ground(B)).        %A3
:- checked success partition(A,B,C,D) => (list(C,num),ground(D)).%A4
:- false calls append(A,B,C) : (list(A,num),list(B,num)).           %A5
```

Assertion A5 has been detected to be false. This indicates a violation of the specification given, which is also flagged by CiaoPP as follows:

```
ERROR: (lns 22-23) false calls assertion:
   :- calls append(A,B,C) : list(A,num),list(B,num)
      Called append(list(^x),[^x|list(^x)],var)
```

The error is now in the call `append(R2,[x|R1],R)` in `qsort` (`x` instead of `X`). After correcting this bug and preprocessing the program again we get:

```
:- checked calls qsort(A,B) : list(A,num).                          %A1
:- check success qsort(A,B)  => sorted_num_list(B).                 %A2
:- checked calls partition(A,B,C,D) : (ground(A),ground(B)).        %A3
:- checked success partition(A,B,C,D) => (list(C,num),ground(D)).%A4
:- checked calls append(A,B,C) : (list(A,num),list(B,num)).         %A5
```

I.e., assertions A1, A3, A4, and A5 have been validated but it was not possible to prove statically assertion A2, which has remained with `check` status, though it has been simplified. On the other hand the analyses used in our session (`regtypes` and `shfr`) do not provide enough information to prove that the output of `qsort` is a *sorted* list of numbers, since this is not a native property of the analyses being used. While this property could be captured by including a more refined domain (such as constrained types), it is interesting to see what happens with the analyses selected for the example.[18]

Dynamic Debugging with Run-time Checks. Assuming that we stay with the analyses selected previously, the following step in the development process is to compile the program obtained above with the "generate run-time checks" option. In the current implementation of CiaoPP we obtain the following code for predicate `qsort` where the new predicate name generated is `qsort_1`:

```
qsort(A,B) :-
        qsort_1(A,B),
        postc([ (qsort(C,D), true, sorted(D)) ], qsort(A,B)).

qsort_1([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R2,[X|R1],R).
qsort_1([],[]).
```

where the code for `partition` and `append` remain the same as there is no other assertion left to check. If we now run the program with run-time checks in order to sort, say, the list `[1,2]`, the Ciao system generates the following error message:

```
?- qsort([1,2],L).
ERROR: for Goal qsort([1,2],[2,1])
Precondition: true  holds, but
Postcondition: sorted_num_list([2,1]) does not.
```

[18] Note that this property, although not provable with the analyses selected, it is however disprovable.

```
L = [2,1] ?
```

Clearly, there is a correctness problem with `qsort`, since `[2,1]` is not the result of sorting `[1,2]` in ascending order. This is a (now, run-time, or *concrete*) incorrectness symptom, which can be used as the starting point of diagnosis. The result of such diagnosis should indicate that the call to `append` (where `R1` and `R2` have been swapped) is the cause of the error and that the right definition of predicate `qsort` is the following:

```
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2), qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).
```

Other CiaoPP Functionalities. In addition to the debugging-related functionality discussed before, CiaoPP includes a number of other capabilities related to the application of analysis results to program optimization, such as program specialization and program parallelization. While most of these functionalities are in general outside our current scope, we will discuss a particular one, abstract (multiple) specialization [3.41, 3.40]. As we will see later, this type of optimization, performed as a source to source transformation of the program, and in which static analysis is instrumental, is indeed relevant in our context.

Program specialization optimizes programs for known values (substitutions) of the input. It is often the case that the set of possible input values is unknown, or this set is infinite. However, a form of specialization can still be performed in such cases by means of abstract interpretation, specialization then being with respect to abstract values, rather than concrete ones. Such abstract values represent of a (possibly infinite) set of concrete values. For example, consider the definition of the property `sorted_num_list/1`, and assume that regular type analysis has produced:

```
:- true pred sorted(A) : list(A,num) => list(A,num).
```

Abstract specialization can use this information to optimize the code into:

```
sorted_num_list([]).
sorted_num_list([_]).
sorted_num_list([X,Y|Z]):- X<Y, sorted_num_list([Y|Z]).
```

which is clearly more efficient because no `number` tests are executed. The optimization above is based on "abstractly executing" the `number` literals to the value `true`, the same notion used in reducing assertions during compile-time checking.

The importance of abstract specialization in our context relies partly in that the availability of the abstract specializer [3.41, 3.40] allows an alternative implementation of the whole framework (also using both compile-time and run-time checking of assertions) by first generating in a naive way a program which performs run-time checking of all assertions and then applying the abstract specializer to this program. The resulting code would be similar to that obtained with the previous approach (first simplifying the assertions in a specialized module and then generating code for those which cannot be statically proved): `checked` assertions will result in run-time tests that are optimized away, `false` assertions will result in run-time tests that are transformed to `error`, etc. However, we have opted for the first alternative because we have found that it is easier for the user to understand things in terms of simplified assertions rather than by looking at the run-time tests which remain in the transformed code.

3.7 Some Practical Hints on Debugging with Assertions

As mentioned before, one of the main features of the preprocessor we present is that assertions are optional and can state partial specifications. The fact that assertions are optional has important consequences on the ease of use and the practicality of the whole approach. An important drawback of many verification systems is the need for a relatively precise specification of the program. Writing such a specification is usually a tedious and not straightforward task. As a result, users in practice often get discouraged and may decide not to use systems which require quite detailed specifications. In contrast, in our framework assertions can be written "on demand", perhaps adding them only for those predicates, program points, and properties that the user wants to check in a given program. Clearly, as more (and more precise) assertions are added to a program, more bugs can potentially be detected automatically. Note that during the process of program development and debugging we will often turn our attention from some parts of the programs to others, and thus the set of assertions may change from one iteration to another.

The fact that assertions are optional obviously raises questions regarding issues such as, for which parts of the program should one write `check` assertions, what kinds of assertions should be used for a given objective, which kind of properties should be used in a given assertion, etc. Many of these questions are still open for research. Nevertheless, we can attempt to provide a few answers.

A point to note is that, from the point of view of their use in debugging, `calls` assertions are conceptually somewhat different from `success` and `comp` assertions. It is not of much use to introduce `success` and `comp` assertions during debugging for predicates which are known to be correct. Introducing `success` and `comp` assertions is in general most useful for *suspect* predicates. On the other hand, introducing `calls` assertions is a good idea even for correct predicates because the fact that a predicate is correct does not guarantee that it is called in the proper way in other parts of the program.

An important remark is that it is usually the case that different parts of the program are perceived by the user as having different levels of reliability [3.23]. For example, in order to detect a bug it is usually good practice to assume that library predicates are correct. For a tool to be successful, we believe that such different levels of reliability should somehow be reflected during the validation/debugging session so that the programmer's attention can concentrate on a particular part of the code. Otherwise the debugging task becomes unrealistic for real programs. This can be achieved in our framework by adding assertions for those predicates that attention is focussed on and by "removing" assertions for others which are no longer under consideration. One very sensible way of doing this is by using modules. Dividing a program into modules allows performing compile-time checking by focusing on a single module, while not judging the code in other modules, of which we are only aware through a high-level description of the imported predicates (i.e., assertions for internal predicates of an imported module are effectively "turned off"). This is the approach used in CiaoPP.

Acknowledgments

The authors would like to thank Saumya Debray and Lee Naish for many interesting discussions on static analysis and debugging, Pawel Pietrzak for his adaptation of John Gallagher's type analysis for CLP(FD), Pedro López for greatly improving the interface to the above mentioned type analysis and making type information available not only at predicate level but also at each point in the program, and Daniel Cabeza for implementing the extended syntactic checking performed by the preprocessor. This work has been partially supported by the European ESPRIT LTR project # 22532 "DiSCiPl" and Spanish CICYT projects TIC99-1151 "EDIPIA" and TIC97-1640-CE.

References

3.1 A. Aggoun and N. Beldiceanu. Overview of the chip compiler system. In *Proc. International Conference on Logic Programming*, pages 775–789. MIT Press, 1991.
3.2 F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.
3.3 J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.
3.4 F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997.
3.5 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.

3.6 F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

3.7 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

3.8 F. Bueno, P. López, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP8/95.0.7.20, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, November 1999.

3.9 F. Bueno, P. López, G. Puebla, M. Hermenegildo, and P. Pietrzak. The CHIP Assertion Preprocessor. Technical Report CLIP1/99.1, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, March 1999. Also as deliverable of the ESPRIT project DISCIPL.

3.10 L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-A. Tärnlund, editor, *Workshop on Logic Programming*, Debrecen, 1980.

3.11 D. Cabeza and M. Hermenegildo. A Modular, Standalone Compiler for Ciao Prolog and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems. To appear*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, 2000.

3.12 D. Cabeza and M. Hermenegildo. The Ciao Module System: A New Module System for Prolog. In *Special Issue on Parallelism and Implementation of (C)LP Systems. To appear*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, 2000.

3.13 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.
The CLIP Group. Program Assertions. The Ciao System Documentation Series – TR CLIP4/97.1, Facultad de Informática, UPM, August 1997.

3.14 M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.

3.15 M. Comini, G. Levi, and G. Vitiello. Abstract debugging of logic programs. In L. Fribourg and F. Turini, editors, *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450, Berlin, 1994. Springer-Verlag.

3.16 The COSYTEC Team. *CHIP System Documentation*, April 1996.

3.17 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.

3.18 P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

3.19 S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

3.20 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.

3.21 W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.
3.22 M. Ducassé. OPIUM - an advanced debugging system. In M. J. Comyn, G.; Fuchs, N.E.; Ratcliffe, editor, *Proceedings of the Second International Logic Programming Summer School on Logic Programming in Action (LPSS'92)*, volume 636 of *LNAI*, pages 303–312, Zurich, Switzerland, September 1992. Springer Verlag.
3.23 M. Ducassé. A pragmatic survey of automated debugging. In Peter A. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, May 1993.
3.24 M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *Journal of Logic Programming*, 19,20:351–384, 1994.
3.25 J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.
3.26 J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.
3.27 M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, 1996.
3.28 M. Hermenegildo. A System for Automatically Generating Documentation for (C)LP Programs. In *Special Issue on Parallelism and Implementation of (C)LP Systems. To appear*, Electronic Notes in Theoretical Computer Science, 2000.
3.29 M. Hermenegildo, F. Bueno, D. Cabeza, M. García de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.
3.30 M. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
3.31 M. Hermenegildo and The CLIP Group. Programming with Global Analysis. In *Proceedings of ILPS'97*, pages 49–52, Cambridge, MA, October 1997. MIT Press. (abstract of invited talk).
3.32 M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.
3.33 M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.
3.34 J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
3.35 N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prenctice Hall, New York, 1993.
3.36 A. Kelly, A. Macdonald, K. Marriott, P. Stuckey, and R. Yap. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International*

Conference and Symposium on Logic Programming, pages 37–51. MIT Press, 1996.

3.37 A. Kelly, K. Marriott, H. Søndergaard, and P.J. Stuckey. A generic object oriented incremental analyser for constraint logic programs. In *Proceedings of the 20th Australasian Computer Science Conference*, pages 92–101, 1997.

3.38 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.

3.39 G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, LNCS. Springer-Verlag, 2000. To appear.

3.40 G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

3.41 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.

3.42 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.

3.43 E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.

3.44 E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Concurrent Prolog: Collected Papers*, pages 211–244, 1987.