Proc. 1996 Joint International Conference and
Symposium on Logic Programming (JICSLP'96),
M. Maher (ed.), The MIT Press, 1996, pp.215-229.

# Diagnosing Non-Well-Moded Concurrent Logic Programs

**Kenta CHO and Kazunori UEDA**
Department of Information and Computer Science
Waseda University
3-4-1, Okubo, Shinjuku-ku, Tokyo 169, Japan
{cho,ueda}@ueda.info.waseda.ac.jp

## Abstract

Strong moding and constraint-based mode analysis are expected to play fundamental roles in debugging concurrent logic/constraint programs as well as in establishing the consistency of communication protocols and in optimization. Mode analysis of Moded Flat GHC is a constraint satisfaction problem with many simple mode constraints, and can be solved efficiently by unification over feature graphs. In practice, however, it is important to be able to analyze non-well-moded programs (programs whose mode constraints are inconsistent) and present plausible "reasons" of inconsistency to the programmers in the absence of mode declarations.

This paper discusses the application of strong moding to systematic and efficient static program debugging. The basic idea, which turned out to work well at least for small programs, is to find a minimal inconsistent subset from an inconsistent set of mode constraints and indicate the symbol( occurrence)s in the program text that imposed those constraints. A bug can be pinpointed better by finding more than one overlapping minimal subset. These ideas can be readily extended to finding multiple bugs at once. For large programs, stratification of predicates narrows search space and produces more intuitive explanations. Stratification plays a fundamental role in introducing mode polymorphism as well.

## 1 Introduction

One of the prominent features of concurrent logic/constraint programming languages is that they allow us to describe interprocess communication with complicated protocols quite easily. Data structures with complicated dataflow, such as streams of messages with reply boxes and streams of streams, can be expressed without any extensions to their simple, basic computation model.

However, most implementations of concurrent logic languages detect fundamental bugs, such as connecting two streams with different communication protocols, as run-time errors. These should ideally be detected statically. Static type systems, if available, may detect some of those bugs, but do not suffice to ensure the consistency of communication protocols used in a whole program. Thus we need a framework of information flow analysis—a mode system. Mode analysis is very useful for program optimization as well as for the static detection of bugs [9].

Most frameworks of mode analysis proposed so far for concurrent and ordinary logic programming languages were based on abstract interpretation. In contrast, the mode system proposed by one of the authors [9] is constraint-based, that is, mode analysis means to solve a system of mode constraints imposed by individual symbols or symbol occurrences in a program. Since the analyzer does not have to trace execution paths, the framework is particularly useful for the analysis of parallel and concurrent languages in which primitive operations are only partially ordered. Another advantage is that it is inherently amenable to the separate analysis of large programs.

Moded Flat GHC [9] takes those advantages of the constraint-based mode system and incorporates it as a language construct rather than just as a framework for program analysis.[1]

An efficient algorithm of mode analysis has already been established for well-moded programs [9]. However, it was not clear how to find, for a non-well-moded program, a plausible "reason" of mode errors efficiently. Since the principal mode of a program is determined by the conjunction of all mode constraints, non-well-modedness means that the conjunction is inconsistent (that is, there are no modes satisfying all the constraints). However, simply reporting that the conjunction of all the mode constraints is inconsistent does not help debugging large programs. The purpose of this paper is to propose practical algorithms that locate the reasons of mode errors as precisely and efficiently as possible.

One may wonder how many of the bugs can be detected by rather simple mode analysis, but our experience has shown that that surprisingly many of them can [10]. Bugs which cannot be identified by mode analysis are likely to be related to problems with algorithms.

## 2 Strong Moding and Mode Analysis

This section outlines the mode system of Moded Flat GHC and the associated mode analysis. Due to space limitations, readers unfamiliar with Moded Flat GHC are referred to [8] for introduction and [9, 10] for technical details and proofs of fundamental properties.

The purpose of the mode system of Moded Flat GHC is to assign to each predicate argument a polarity structure that defines the direction of information flow of each part of data structures. The polarity structure is computed by mode analysis so that each part of data structures will be instantiated cooperatively, namely by *exactly one* goal.

A mode in our mode system is a function from the set of *paths* for specifying each "part" of data structures to the two-valued codomain $\{in, out\}$. Paths here are strings of pairs, of the form $\langle symbol, arg\rangle$, of predicate/function symbols and argument positions. Formally, the set $P_{Term}$ of paths for terms and the set $P_{Atom}$ of paths for atomic formulae are defined

[1]Strong moding can be incorporated in ordinary logic programming languages as well [7], but it is particularly important in concurrent logic programming because most concurrent logic programs have fixed dataflow and make more restricted use of unification. However, the diagnosis technique proposed in this paper is quite general.

using disjoint union as:

$$P_{Term} = (\sum_{f \in Fun} N_f)^* \; , \; P_{Atom} = (\sum_{p \in Pred} N_p) \times P_{Term} \quad ,$$

where $\mathit{Fun}/\mathit{Atom}$ are the sets of function/predicates symbols, and $N_f/N_p$ are the sets of possible argument positions (numbered from 1) for the symbols $f/p$.

The purpose of the mode system is to find a mode $m : P_{Atom} \to \{\mathit{in}, \mathit{out}\}$ under which every piece of communication is cooperative. Such a mode is called a *well-moding*. Intuitively, *in* means the inlet of information and *out* means the outlet of information.

Well-modings can be computed by solving mode constraints imposed by (i) the occurrences of function symbols and (ii) the variable symbols in a program and an initial goal clause. A program does not usually define a unique well-moding but has many of them. So the purpose of mode analysis is to compute the set of all well-modings in the form of a *principal* (i.e., most general) mode. Principal modes can be expressed naturally by mode graphs, as described later in this section.

Given a mode $m$, we define a *submode* $m/p$, namely $m$ viewed at the path $p$, as a function satisfying $(m/p)(q) = m(pq)$. We also define $\mathit{IN}$ and $\mathit{OUT}$ as submodes always returning *in* and *out*, respectively. An overline '$\overline{\phantom{x}}$' inverts the polarity of a mode, a submode, or a mode value.

A Flat GHC program is a set of clauses of the form $h$ `:-` $G$ `|` $B$, where $h$ is an atomic formula and $G$ and $B$ are multisets of atomic formulae. Constraints imposed by a clause $h$ `:-` $G$ `|` $B$ are summarized in Figure 1, where $\tilde{a}(p)$ means a symbol occurring at the path $p$ in an atomic formula $a$, $\mathit{Var}$ is the set of variable symbols, and $\mathit{Term}$ is the set of terms defined over $\mathit{Fun}$ and $\mathit{Var}$. Rule (BU) numbers unification body goals because the mode system allows different body unification goals to have different modes. This is a special case of mode polymorphism we will discuss in Section 6.

As an example, consider the following program for stream merging:

```
m([],Y,Z) :- true | Z=₁Y.
m(X,[],Z) :- true | Z=₂X.
m([A|X],Y,Z0) :- true | Z0=₃[A|Z], m(X,Y,Z).
m(X,[A|Y],Z0) :- true | Z0=₄[A|Z], m(X,Y,Z).
```

The third clause, for example, imposes the following eight constraints ("`.`" stands for the constructor of non-empty lists):

$$\begin{aligned}
m(\langle \mathtt{m}, 1\rangle) &= \mathit{in} && \text{by (HF) applied to "}\mathtt{.}\text{"}\\
m/\langle \mathtt{=}_3, 1\rangle &= \overline{m/\langle \mathtt{=}_3, 2\rangle} && \text{by (BU) applied to } \mathtt{=}_3\\
m(\langle \mathtt{=}_3, 2\rangle) &= \mathit{in} && \text{by (BF) applied to "}\mathtt{.}\text{"}\\
m/\langle \mathtt{m}, 1\rangle\langle \mathtt{.}, 1\rangle &= m/\langle \mathtt{=}_3, 2\rangle\langle \mathtt{.}, 1\rangle && \text{by (BV) applied to } \mathtt{A}\\
m/\langle \mathtt{m}, 1\rangle\langle \mathtt{.}, 2\rangle &= m/\langle \mathtt{m}, 1\rangle && \text{by (BV) applied to } \mathtt{X}\\
m/\langle \mathtt{m}, 2\rangle &= m/\langle \mathtt{m}, 2\rangle && \text{by (BV) applied to } \mathtt{Y}\\
m/\langle \mathtt{m}, 3\rangle &= \overline{m/\langle \mathtt{=}_3, 1\rangle} && \text{by (BV) applied to } \mathtt{Z0}\\
m/\langle \mathtt{=}_3, 2\rangle\langle \mathtt{.}, 2\rangle &= \overline{m/\langle \mathtt{m}, 3\rangle} && \text{by (BV) applied to } \mathtt{Z}
\end{aligned}$$

(HF) $\forall p \in P_{Atom}(\widetilde{h}(p) \in Fun \Rightarrow m(p) = in)$
(If a function symbol occurs at $p$ in $h$, $m(p) = in$.)

(HV) $\forall p \in P_{Atom}(\widetilde{h}(p) \in Var \wedge \exists p' \neq p(\widetilde{h}(p) = \widetilde{h}(p')) \Rightarrow m/p = IN)$
(If the symbol at $p$ in $h$ is a variable occurring elsewhere in $h$, $m/p = IN$.)

(GV) $\forall p, p' \in P_{Atom}\, \forall a \in G(\widetilde{h}(p) \in Var \wedge \widetilde{h}(p) = \widetilde{a}(p') \Rightarrow \forall q \in P_{Term}(m(p'q) = in \Rightarrow m(pq) = in))$
(If the same variable occurs both at $p$ in $h$ and at $p'$ in $G$, then $m(pq) = in$ if the path $m(p'q)$ is examined in a guard goal.)

(BU) $\forall k > 0\, \forall t_1, t_2 \in Term((t_1 =_k t_2) \in B \Rightarrow m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle})$
(The two arguments of a unification body goal have opposite submodes.)

(BF) $\forall p \in P_{Atom} \forall a \in B(\widetilde{a}(p) \in Fun \Rightarrow m(p) = in)$
(If a function symbol occurs at $p$ in a body goal, $m(p) = in$.)

(BV) Let $v$ be a variable occurring exactly $n\,(\geq 1)$ times in $h$ and $B$ at $p_1, \ldots, p_n$, of which the occurrences in $h$ are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

$$\begin{cases} \mathcal{R}(\{m/p_1, \ldots, m/p_n\}), & \text{if } k = 0; \\ \mathcal{R}(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}), & \text{if } k > 0; \end{cases}$$

where the unary predicate $\mathcal{R}$ over finite *multisets* of submodes represents "cooperative communication" between paths and can be defined as

$$\mathcal{R}(S) \overset{\text{def}}{=} \forall q \in P_{Term}\, \exists s \in S(s(q) = out\ \wedge\ \forall s' \in S \backslash \{s\}\, (s'(q) = in)).$$

Figure 1: Constraints imposed by a clause $h$ :- $G$ | $B$

From the entire definition, we obtain 24 constraints. Elimination of the constraints on $=_k$, however, leaves only four constraints:

$$m(\langle \mathtt{m}, 1\rangle) = in, \qquad m/\langle \mathtt{m}, 1\rangle\langle\, .\,, 2\rangle = m/\langle \mathtt{m}, 1\rangle,$$
$$m/\langle \mathtt{m}, 2\rangle = m/\langle \mathtt{m}, 1\rangle, \qquad m/\langle \mathtt{m}, 3\rangle = \overline{m/\langle \mathtt{m}, 1\rangle}.$$

We could regard the above set of constraints itself as representing the principal mode of the program, but the principal mode can be represented more explicitly in terms of a mode graph (Figure 2). Mode graphs are a kind of features graphs (feature structures with cycles) [1] in which

1. paths represent paths in $P_{Atom}$,
2. the node corresponding to the path $p$ represents the value $m(p)$,
3. arcs are labelled with the pair $\langle symbol, arg\rangle$ of predicate/function symbols and argument positions, and may have "negative signs" (denoted "•" in Figure 2) that invert the interpretation of the mode values of the paths beyond those arcs, and

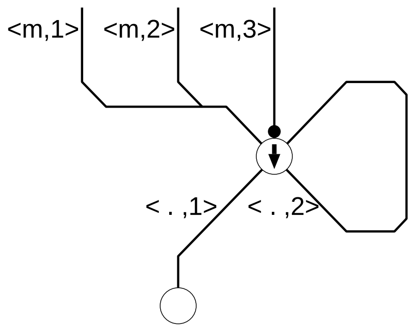

Figure 2: The mode graph of a stream merging program

4. binary constraints of the forms $m/p_1 = m/p_2$ and $m/p_1 = \overline{m/p_2}$ are represented by the sharing of nodes.

Mode analysis proceeds by merging many simple mode graphs representing individual mode constraints. Thus its decidability is guaranteed by the decidability of the unification algorithm for feature graphs. The principal mode of a well-moded program, represented as a mode graph, is uniquely determined, as long as all the mode constraints imposed by the program are unary (i.e., constraint on the mode value of, or the submode at, a particular path) or binary (i.e., constraint between the submodes at two particular paths).

Rule (GV) in Figure 1 contains a conditional mode constraint. However, we assume that guard goals are calls to built-in predicates whose mode graphs have been obtained beforehand. Thus the constraints actually imposed by Rule (GV) will be of the unary form $m(p) = in$ or $m/p = IN$. The rule will become simpler by allowing polymorphic guard goals, as will be discussed in Section 6.

Rule (BV) may impose constraints between three or more constraints, which cannot be represented as mode graphs by themselves. However, by *delaying* them, most of them can be reduced to unary/binary ones by other constraints [10]. Theoretically, some non-binary constraints may remain unreduced, for which it is most practical to let programmers declare the submodes of relevant paths.

The cost of mode analysis is almost proportional to the size $n$ of the program and to the size $d$ of the subgraph of the entire mode graph rooted at each predicate argument [9]. The size $d$ reflects the complexity of communication protocols used in the program. To be precise, the time complexity is $O(nd \cdot \alpha(n))$, where $\alpha$ is the inverse of the Ackermann function.

We have analyzed various GHC/KL1 programs including the mode analyzer itself [10]. We have observed that, although larger programs have larger mode graphs because they use more predicate symbols, the value of $d$ does not become so large (say several tens of nodes) even for programs using quite complicated communication protocols. Thus we expect that the mode graphs of very large programs are, in general, wide and shallow, which is to say most nodes can be reachable within several steps from the root.

## 3 Non-Well-Moded Programs

Programs that do not have well-modings are called non-well-moded. Since Moded Flat GHC programs must observe the principle of cooperative communication, non-well-modedness indicates that the communication protocols specified by the program are faulty. For instance, the following quicksort program is non-well-moded:

```
qsort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
qsort([],    Ys0,Ys ) :- true | Ys =$_1$ Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
   part(X,Xs,S,L), qsort(S,Ys0,Ys1), Ys2 =$_2$ [X|Ys1], qsort(L,Ys2,Ys3).
   (the unification goal should have been Ys1 =$_2$ [X|Ys2])
```

The first clause imposes the constraint (among others)

(1) $m(\langle \texttt{qsort}, 3\rangle) = \mathit{in}$ by (BF) applied to "`[]`",

while the second clause imposes

(2) $m/\langle \texttt{=}_1, 1\rangle = m/\langle \texttt{qsort}, 3\rangle$ by (BV) applied to `Ys`,
(3) $m/\langle \texttt{=}_1, 2\rangle = \overline{m/\langle \texttt{=}_1, 1\rangle}$ by (BU) applied to $\texttt{=}_1$,
(4) $m/\langle \texttt{qsort}, 2\rangle = m/\langle \texttt{=}_1, 2\rangle$ by (BV) applied to `Ys0`.

The third clause is first normalized to

```
qsort([X|Xs],Ys0,Ys3) :- true |
   part(X,Xs,S,L), qsort(S,Ys0,Ys1), qsort(L,[X|Ys1],Ys3).
```

and imposes

(5) $m/\langle \texttt{qsort}, 2\rangle = \mathit{in}$ by (BF) applied to"`|`".

The set of those five constraints are inconsistent. Constraints (1)–(4) together entail $m(\langle \texttt{qsort}, 2\rangle) = \mathit{out}$, which is clearly inconsistent with Constraint (5).

We consider explaining the reasons of mode errors in terms of minimal inconsistent subsets of the set of mode constraints, because such subsets will be useful for locating errors. If we find multiple inconsistent subsets that are disjoint, they are considered as indicating different bugs in the program. Thus the technique can be used for locating multiple bugs at once.

Let us get back to the quicksort example. Observe that all proper subsets of Constraints (1)–(5) are consistent. Hence the five constraints form a minimal inconsistent subset. Since the quicksort program imposes 53 constraints in total, including constraints from the predicate `part`, we have succeeded in finding an adequately small subset.

Once a minimal inconsistent subset is found, how can one pinpoint a bug? It is reasonable in a moded framework to assume that programmers have *intended modes* of their programs (though not declared explicitly). In the above example, the programmer should be able to find that Constraint (5) is wrong because the second argument of `qsort` is intended to return the result of sorting. The analyzer tells what symbol occurrence in what clause imposes that constraint, which is the exact location of the bug.

A programmer may not always find it easy to tell whether each single constraint in a minimal subset conforms to the intended mode. However, the analyzer can present various consequences of a minimal subset of constraints as follows: Suppose the minimal subset $S$ contains constraints from $n$ ($> 1$) clauses. Then, it can be divided into $n$ disjoint subsets $S_1, \ldots, S_n$ based on what clauses imposed what constraints. Because $S$ is minimal, each of $S_1, \ldots, S_n$ is consistent. So we can form a mode graph for each $S_i$ to make the consequences entailed by $S_i$ explicit and find what clause contains a bug.

The mode constraints imposed by a program usually have redundancy. That is, a single bug could be explained by many possible minimal inconsistent subsets. However, a subset corresponding to a local portion of a program is likely to be a better explanation than a subset corresponding to a larger or scattered portion, because it reflects the program structure better and facilitates debugging. So, after describing basic algorithms in Section 4, we consider in Section 5 how to divide programs into layers based on the structure of process definitions and search minimal subsets locally within each process definition.

## 4 Finding Minimal Subsets

We consider various algorithms for finding a minimal inconsistent subset of a inconsistent set of mode constraints $C = \{c_1, \ldots, c_n\}$. This section presents simple algorithms for finding a single inconsistent subset and extend them to find multiple disjoint subsets.

### 4.1 The Basic Algorithms

Let $C = \{c_1, \ldots, c_n\}$ be an inconsistent set of constraints. Algorithm 1 below finds a single minimal inconsistent subset from $C$. In the algorithm, the merging of constraint sets and the checking of consistency are realized as the unification of mode graphs and the checking of its success/failure. Although the algorithm is quite general, its efficiency hinges upon the fact that there is a pair of efficient algorithms for computing the union of constraint sets and checking its consistency.

- **Algorithm 1**

```
$S \leftarrow \{\}$;
**while** $S$ is consistent **do**
    $D \leftarrow S$; $i \leftarrow 0$;
    **while** $D$ is consistent **do**
        $i \leftarrow i + 1$; $D \leftarrow D \cup \{c_i\}$
    **end**;
    $S \leftarrow S \cup \{c_i\}$
**end**
```

The set $S$ thus obtained is a minimal inconsistent subset. To see why, let $S_j/D_j/i_j$ be the values of $S/D/i$ at the end of the $j$th iteration of the outer loop, and $k$ be the size of $S$. It suffices to show that the set $S_k \setminus \{c_{i_j}\} = \{c_{i_1}, \ldots, c_{i_{j-1}}, c_{i_{j+1}}, \ldots, c_{i_k}\}$ is consistent for any $i_j$, but it is easy to see that

- $D_j \setminus \{c_{i_j}\}$ is consistent (because the inner loop was not exited when $D = D_j \setminus \{c_{i_j}\}$), and
- $S_k \setminus \{c_{i_j}\} = \{c_{i_1}, \ldots, c_{i_{j-1}}, c_{i_{j+1}}, \ldots, c_{i_k}\} \subseteq D_j \setminus \{c_{i_j}\}$, because
  - $D_j \setminus \{c_{i_j}\} = S_{j-1} \cup \{c_1, c_2, \ldots, c_{i_j-1}\}$,
  - $S_{j-1} = \{c_{i_1}, c_{i_2}, \ldots, c_{i_{j-1}}\}$, and
  - $\{c_{i_{j+1}}, \ldots, c_{i_k}\} \subseteq \{c_1, c_2, \ldots, c_{i_j-1}\}$ (because $i_1 > i_2 > \ldots > i_k$).

Hence all proper subsets of $S$ are consistent.

Now we consider the complexity of the above algorithm. As explained in Section 2, it takes $\mathrm{O}(nd \cdot \alpha(n))$ time to merge $n$ mode constraints. The time complexity of finding a minimal subset with $k$ elements out of $n$ mode constraints is $\mathrm{O}(nkd \cdot \alpha(n))$, because

- in each iteration, we must merge at most $n$ constraints until inconsistency arises, and
- it takes $k$ iterations until a minimal subset with $k$ elements is obtained.

Usually, $k$ is a small value independent of the program size, as we will see in Section 7.

A variant of the above algorithm will compute a better minimal subset. Let $C = \{c_1, \ldots, c_n\}$ be such that $i < j$ implies that the symbol (occurrence) imposing $c_i$ occurs textually before the symbol (occurrence) imposing $c_j$. Then it is likely that a minimal subset can be formed from a rather small range of the sequence $c_1, \ldots, c_n$, and such a local subset is considered a good explanation. If this is the case, scanning $S$ in alternate directions will be more efficient and compute a better solution:

- **Algorithm 1′**

```
    $S \leftarrow \{\}$; $i \leftarrow 0$; $j \leftarrow 1$;
    while $S$ is consistent do
        $D \leftarrow S$;
        while $D$ is consistent do
            $i \leftarrow i + j$; $D \leftarrow D \cup \{c_i\}$
        end;
        $S \leftarrow S \cup \{c_i\}$; $j \leftarrow -j$
    end
```

## 4.2 Finding Multiple Independent Minimal Subsets

Algorithms 1 and 1′ compute a single minimal inconsistent subset $S$ of $C$. To compute multiple, independent minimal subsets, we can simply re-apply the algorithms after removing the elements of $S$ from $C$. This enables the analyzer to detect as many independent bugs as possible at once. Note that Algorithm 2 below uses a self-contradictory constraint as a sentinel.

- **Algorithm 2**

```
$c_0 \leftarrow$ false;
**while** true **do**
    let $c_1, \ldots, c_m$ be the elements of $C$;
    $i \leftarrow m + 1$; $j \leftarrow -1$; $S \leftarrow \{\}$;
    **while** $S$ is consistent **do**
        $D \leftarrow S$;
        **while** $D$ is consistent **do**
            $i \leftarrow i + j$; $D \leftarrow D \cup \{c_i\}$
        **end**;
        $S \leftarrow S \cup \{c_i\}$; $j \leftarrow -j$
    **end**;
    **if** $i = 0$ **then exit**
    **else**
        report($S$); $C \leftarrow C \backslash S$
    **fi**
**end**
```

## 5 Diagnosing Stratified Programs

Algorithms in Section 4 find minimal subsets from the entire set of constraints without reference to the logical structure of the programs to be analyzed. However, the set of constraints can be very large. If we divide the set of constraints taking the problem domain (= program analysis) into account and analyze the obtained subsets separately, we may be able to reduce the amount of computation and obtain more useful information for debugging.

### 5.1 Call Graphs and Process Graphs

When dividing Flat GHC programs according to their logical structures, clauses defining a concurrent process by means of self or mutual recursion can be considered to form a *process definition*.

A program defines a directed graph, called a *call graph*, that describes the caller-callee relationship between predicates. A call graph is a directed graph such that each node $v$ corresponds to a Flat GHC predicate, and an arc $e$ from a node $v$ to a node $v'$ means that the predicate $v$ calls the predicate $v'$ directly from a clause body.

The strongly connected components of a call graph exactly correspond to process definitions in the above sense. Processes defined by mutual recursion are naturally recognized in this way. A non-recursive predicate that spawns one or more subprocesses is regarded as a process by itself.

It is well-known that division into strongly connected components is uniquely determined and can be done in $O(n + a)$ time, where $n$ is the number of nodes and $a$ is the number of arcs in the graph.

Division into strongly connected components is regarded as the division of graph nodes into equivalence classes. The quotient graph obtained by contracting the arcs inside strongly connected components is called a *process graph.* A process graph is a dag representing the dependency relation between processes. Henceforth we confuse a node of a process graph with the process definition represented by the node.

## 5.2 Program Stratification

Since the process graph $G$ of a program is acyclic, the partial order defined by $G$ can be used for stratifying the program. We define the layer number $L(v)$ of the node $v$ as:

$$L(v) = \max(\{L(v') \mid v' \in Adj^{+}(v)\} \cup \{0\}) + 1 \ ,$$

where $Adj^{+}(v)$ means the set of destination nodes of the arcs from $v$. Note that the above definition assigns 1 to nodes without outgoing arcs.

## 5.3 Finding Relative Minimal Subsets

Bugs of stratified programs can be classified into (1) those within each layer and (2) those across layers. Since bugs of the first kind can be found simply by checking each node of a process graph independently, we consider how to deal with bugs of the latter kind, assuming that each process definition is well-moded.

How to find a minimal subset from a stratified program depends on how we consider non-well-modedness across layers. We adopt bottom-up analysis, that is, we choose to check process definitions from lower layers (those with smaller numbers). Bottom-up analysis lends itself to the analysis of large programs that may use existing program libraries.

Suppose bottom-up analysis has found an inconsistency in an attempt to merge constraints from the $k$th layer and those from lower layers. Since each process definition in the $k$th layer is consistent by assumption and different process definitions in the same layer are independent, the $k$th layer itself is consistent. Hence the reason of inconsistency can be attributed to either (or both) of the following:

- The $k$th layer wrongly uses the lower layers.
- The lower layers, though well-moded, have an unintended principal mode.

It is rather difficult to tell which, but a reasonable solution is to ask the programmer to check the $k$th layer first before suspecting lower layers. This is reasonable because a concise explanation should be considered first. Locating bugs inside well-moded layers is somewhat beyond the principal scope of mode analysis, though their mode graphs will provide useful information.

Bottom-up analysis considerably limits search space by finding from a process definition a minimal subset of constraints that are inconsistent with the set $B$ of constraints from lower layers. Such a subset is called a *minimal subset relative to $B$*. In the following algorithm, $C(v)$ initially holds the set of constraints imposed by the node $v$ of the process graph.

- **Algorithm 3**

```
for $k \leftarrow 1$ to the highest layer number do
    for each $v$ in $\{v \mid L(v) = k\}$ do
        $B \leftarrow \bigcup_{v' \in Adj^+(v)} C(v')$;
        apply Algorithm $2'$ (shown below);
        $C(v) \leftarrow B \cup C(v)$
    end
end
```

Algorithm $2'$ reports and removes minimal inconsistent subsets of the set $C(v)$ of constraints relative to $B$:

- **Algorithm $2'$**

```
$c_0 \leftarrow$ false;
while true do
    let $c_1, \ldots, c_m$ be the elements of $C(v)$;
    $i \leftarrow m + 1$; $j \leftarrow -1$; $S \leftarrow B$;
    while $S$ is consistent do
        $D \leftarrow S$;
        while $D$ is consistent do
            $i \leftarrow i + j$; $D \leftarrow D \cup \{c_i\}$
        end;
        $S \leftarrow S \cup \{c_i\}$; $j \leftarrow -j$
    end;
    if $i = 0$ then exit
    else
        $S \leftarrow S \backslash B$;
        report($S$); $C(v) \leftarrow C(v) \backslash S$
    fi
end
```

In an actual implementation, $S$ and $D$ are represented during iterations as mode graphs which are destructively updated by the set union operations. The results to be reported should be represented again by the set of constraints, but this can be obtained efficiently by recording what $c_i$'s have been added to $S$ by the assignment $S \leftarrow S \cup \{c_i\}$.

## 6 Stratification and Mode Polymorphism

When two or more processes share a process definition in a lower layer, Algorithm 3 may cause a problem.

Consider a program that uses a "generic" (or polymorphic) predicate such as stream merging (Section 2) in various modes. When such a predicate is called from different places, the process graph will contain a shared node. Suppose a process definition at the node $v$ and another definition at $v'$ use a predicate $p$ polymorphically. Then the analyses of the node $v$ and of $v'$ will succeed because they are independent, but the analysis of a higher-level node that uses both $v$ and $v'$ will detect the inconsistent use of $p$ as an error.

However, we can regard predicates at lower layers as polymorphic when called from higher layers. To allow stratification-based polymorphism, we need to create a copy of the mode graph of a polymorphic predicate for each call to that predicate. This can be achieved by indexing each polymorphic call (as we have done for unification goals) and creating a copy of the mode graph for each polymorphic call, modifying their paths according to the indices.

So, we assume that

1. for each polymorphic predicate $p$, the preprocessing phase numbers all calls to $p$ from higher layers from 1 upwards, and
2. the first element of each path in $P_{Atom}$ is of the form $\langle p_s, i\rangle$, where $s$ is a sequence of natural numbers and can be omitted if empty.

Also, let $C_k(v)$ be a modified copy of the mode constraints $C(v)$ such that the first elements of the paths are changed from the form $\langle p_s, l\rangle$ to $\langle p_{sk}, l\rangle$.

- **Algorithm 3′**

  The same as Algorithm 3, except that the assignment $B \leftarrow \bigcup_{v' \in Adj^+(v)} C(v')$ is replaced by:

```
    $B \leftarrow \{\}$;
    for each indexed (i.e., polymorphic) body goal $g$ in $v$ do
        let $k$ be the index of $g$;
        let $v'$ be the node defining the polymorphic predicate;
        $B \leftarrow B \cup C_k(v')$
    end
```

Although we have focused on the stratification of predicates called from clause bodies, the same idea could be applied to test predicates called from guards. This is useful for introducing polymorphism to test predicates, under which Rule (GV) in Figure 1 does not have to use implication any more to avoid the "back propagation" of mode constraints to generic guard predicates.

## 7 Experiments

We have made some initial experiments to see how the basic algorithms help bug location and how large minimal subsets can be.

First, we applied Algorithm 1′ to 20 erroneous programs, each containing a single near-miss bug such as

1. "tell" unification specified in a clause head (as in Prolog) rather than in a body,
2. misspelling of a variable name, and
3. wrong order of arguments.

All those bugs will impose constraints inconsistent with those from correct clauses.

Sixteen of the 20 programs we used were small and imposed less than 100 constraints each, while the remaining four programs imposed about 500 constraints each. The sizes of the minimal inconsistent subsets varied from 2 to 8, with the average being 3.75. The sizes of minimal subsets were independent of the total number of constraints. Thus we have ascertained that the parameter $k$ in the complexity measure in Section 4.1 is a rather small constant.

We have also ascertained that multiple bugs can be detected at once if they are not too close to each other. However, since our algorithm removes some correct constraints together with incorrect constraints when finding the first bug, it is possible that the second bug does not cause inconsistency any more. Fortunately, this will not happen so often because the mode constraints imposed by a program usually contain redundancy and the number of removed correct constraints is usually small.

We have not yet analyzed very large programs, but thanks to the constraint-based approach, large programs can (and will) be analyzed in smaller pieces. Stratification will automatically divide a program into pieces, too. Thus we can expect that our positive results will apply to larger programs quite well.

It would be unrealistic to search for *all* minimal inconsistent subsets covering a single bug because it requires much more computation. However, it will be less unrealistic to compute *several* inconsistent subsets which share some constraint. If the program contains a single bug, a constraint shared by all minimal subsets is likely to indicate the bug. For instance, suppose we wrote the stream merging program as:

```
m([],Y,Z) :- true | Z=₁Y.
m(X,[],Z) :- true | Z=₂X.
m([A|X],Y,Z0) :- true | Z0=₃[A|Z], m(X,Y,Z0).
(the final goal should have been m(X,Y,Z))
m(X,[A|Y],Z0) :- true | Z0=₄[A|Z], m(X,Y,Z).
```

The mode analyzer first normalized the program, converting the third clause to

```
m([A|X],Y,Z0) :- true | Z0=₃[A|Z], m(X,Y,[A|Z]).
```

and then found that it had at least four minimal subsets (with 5, 5, 4, and 4 elements). The only constraint included in all of those subsets was $m(\langle \mathtt{m}, 3\rangle) = \mathit{in}$, which was imposed by Rule (BF) applied to the list constructor occurring in the third argument of the recursive goal of the (normalized) third clause. Thus we succeeded in pinpointing the exact location of the bug in this case. It is a subject of future work how to compute a sufficient number of overlapping subsets efficiently to pinpoint a bug.

## 8 Related Work

As mentioned in Section 1, most previous work on the mode analysis of (concurrent) logic languages was based on abstract interpretation, and focused mainly on the reasoning of program properties assuming that the programs were correct. In contrast, constraint-based mode analysis can be used for diagnosis as well as optimization by assuming that correct programs are well-moded.

Concurrent logic languages Doc [2] and Janus [5] let programmers distinguish between input and output occurrences using annotations. These annotations can be regarded as mode declarations, the consistency of which needs to be checked statically or dynamically. So the technique proposed in this paper applies also to those languages. The purpose of the mode system of PARLOG is quite different, as discussed in [9].

Somogyi [6] proposed another framework of strong moding independently and studied its implications in depth. His framework shares some features with ours, such as the principle of cooperative communication and the capability of dealing with bidirectional communication. An advantage of our constraint-based framework is that, besides being simple, it provides a unified framework for mode declaration, mode checking and mode inference. This makes it realistic to analyze existing programs and still enables programmers to declare intended modes that can be used as correct mode constraints in finding minimal subsets.

Mercury [7] is another recent strongly moded language. Being a purely declarative logic language, however, its mode system is very different from the mode system of Moded Flat GHC; the former deals with the change of instantiatedness, which is a temporal property, while the latter deals with polarity, which is a non-temporal property [10].

Chen et al. [3] proposed an algorithm for finding maximal unifiable subsets and minimal non-unifiable subsets of a set of equations. They use hypergraph structures that record the reasons of (non-)unifiability during unification. Our algorithms use mode graphs that do not retain reason information and reconstruct them repeatedly to find minimal subsets. Although this simple approach turned out to work quite well, it is a subject of future work to compare our approach with the hypergraph approach.

Analysis of malfunctioning systems based on their intended logical specification has been studied in the field of artificial intelligence [4] and known as model-based diagnosis. Model-based diagnosis has similarities with our work in the ability of searching minimal explanations and multiple faults. However, the purpose of model-based diagnosis is to analyze the differences between intended and observed behaviors. Our mode system does *not* require that the intended behavior of a program be given as mode declarations, and still locates bugs quite well.

## 9 Conclusions

We have proposed algorithms for diagnosing non-well-moded concurrent logic programs based on the searching of minimal inconsistent subsets of mode constraints. Once minimal subsets are found, it is straightforward for the system to indicate suspected symbol( occurrence)s in the program and/or to show logical consequences a (consistent) subset of the minimal inconsistent subsets entails. We have also shown how we can obtain "good" (i.e., local) explanations of mode errors by dividing programs based on their logical structures. All these techniques are very systematic for static error analysis and are efficient as well.

It is not realistic or helpful to search all minimal inconsistent subsets, but it might be reasonable to find several of them because, if some mode constraint is shared by all the minimal subsets, it is likely to indicate the exact location of a bug. This means to take advantage of the redundancy of mode constraints to guess the exact location of a bug. By pinpointing erroneous constraints this way, the ability of detecting multiple bugs will be improved further.

**Acknowledgments.** The authors are indebted to the anonymous referees for their helpful comments.

## References

[1] Aït-Kaci, H. and Nasr, R., LOGIN: A Logic Programming Language with Built-In Inheritance. *J. Logic Programming*, Vol. 3, No. 3 (1986), pp. 185–215.

[2] Hirata, M., Programming Language Doc and Its Self-Description or, $X = X$ is Considered Harmful. In *Proc. 3rd Conf. of Japan Society of Software Science and Technology*, 1986, pp. 69–72.

[3] Chen, T. Y., Lassez, J.-L. and Port, G. S., Maximal Unifiable Subsets and Minimal Non-unifiable Subsets. *New Generation Computing*, Vol. 4 (1986), pp. 133–152.

[4] Reiter, R., A Theory of Diagnosis from First Principles. Artificial Intelligence, Vol. 32 (1987), pp. 57–95.

[5] Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conf. on Logic Programming*, Debray, S. and Hermenegildo, M. (eds.), MIT Press, 1990, pp. 431–446.

[6] Somogyi, Z., A Parallel Logic Programming System Based on Strong and Precise Modes. Ph. D. thesis, Tech. Report 89/4, Dept. of Computer Science, Univ. of Melbourne, Melbourne, Australia, 1989.

[7] Somogyi, Z., Henderson, F. and Conway, T., Mercury: An Efficient Purely Declarative Logic Programming Language. Proc. Australian Computer Science Conference, Glenelg, Australia, 1995, pp. 499–512.

[8] Ueda, K., I/O Mode Analysis in Concurrent Logic Programming. In *Theory and Practice of Parallel Programming*, LNCS 907, Springer, 1995, pp. 356–368.

[9] Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.

[10] Ueda, K., Experiences with Strong Moding in Concurrent Logic/ Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems*, LNCS 1068, Springer, 1996, pp. 134–153.