Analysis of Parallel Programs

Leonardo Mendonça de Moura
e-mail: moura@les.inf.puc-rio.br

Carlos José Pereira de Lucena
e-mail: lucena@inf.puc-rio.br

Edward Hermann Haeusler
e-mail: hermann@inf.puc-rio.br

PUC-RioInf.MCC06/00 February, 2000

Abstract: Parallel implementations are of growing interest, as they are more and more supported by modern hardware. However, despite its importance there is little work on classical data flow analysis for parallel languages. Probably, the reason for such deficiency is that a naive adaptation of classical data flow analysis fails and the straightforward correct adaptation needs an unacceptable effort, which is caused by considering all possible execution traces of a parallel program. Therefore, we propose an approach to implement program analyzers for parallel languages based on theories of Abstract Interpretation, Model Checking and Constraint Solving. We also consider features such as dynamic memory allocation (aliases) and dynamic process creation, which are often ignored by other approaches.

Keywords: Program Analysis, Parallel & Concurrent Languages.

Resumo: Implementações paralelas estão ganhando interesse a medida que uma quantidade hardware cada vez maior é suportada. Entretanto, apesar de sua importância, pouco foi feito na área de análise de porgramas paralelos. Provavelmente, a principal razão está relacionada as deficiências da abordagem tradicional de análise, e a dificuldade em realizar-se uma adaptação correta, devido a necessidade de considerar-se todos os possíveis traces de execução de um programa. Portanto, nós estamos propondo uma nova abordagem para a implementação de analisadores de programa para linguagens paralelas, utilizando as teorias de Interpretação Abstrata, Model Checking, e Constraint Solving.

Palavras-chave: Análise de Programas, Linguagens Concorrentes & Paralelas.

$$
\begin{aligned}
stmt ::= \; & stmt \;\mathbf{;}\; stmt \mid stmt \parallel stmt \mid id := expr \mid \\
& \mathbf{if}\ expr\ \mathbf{then}\ stmt\ \mathbf{else}\ stmt \mid \\
& \mathbf{while}\ expr\ \mathbf{do}\ stmt \\
& \mathbf{wait}\ expr \\
expr ::= \; & \ldots
\end{aligned}
$$

Figure 1: Simple parallel imperative language abstract syntax tree

1 Introduction

The primary objective of program analysis is to collect information about run-time behavior of a program. Compilers usually use such kind of information to apply optimizations such as loop-invariant code motion, common-subexpression elimination and dead code elimination [11]. The produced information can also help the user to construct, understand and debug programs. Analysis information is also useful in program transformations, partial evaluation, and re-engineering activities.

"Classical" data flow analysis is based on two things: an abstraction of the information needed, and the propagation of the information along the statements of the source program [11]. The abstract information is usually represented by the elements of a semi-lattice. The effect of a single program statement is then a function over these semi-lattice values. The propagation is modeled by applying these functions in the order given by the statements. Unfortunately, a naive adaptation of the "classical" approach fails and the straightforward correct adaptation needs an unacceptable effort, which is caused by considering all possible execution traces of a parallel program.

Extensive work has been done on verifying parallel programs by using techniques such as Model Checking [3, 10], but little work has been done to optimize them. Therefore, we propose an approach which relies on theories of Abstract Interpretation [4], Model Checking [3, 10] and Constraint Solving [7].

1.1 The Simple Imperative Parallel Language

All examples described in this paper are based on a parallel imperative language. Initially, we restrict it to a minimal set of features which are sufficient to describe the essence of our approach. We gradually add new features which complicate the design of program analyzers. Figure 1 contains the abstract syntax tree of our "initial" parallel imperative language. The statement $s_1 \parallel s_2$ represents the parallel composition of the statements s_1 and s_2, *i.e.* a process is created to execute each statement. The execution of $s_1 \parallel s_2$ terminates precisely when both s_1 and s_2 terminates. The order in which the actions occur in each s_1 and s_2, respectively, is preserved in the execution of $s_1 \parallel s_2$. The statement **wait**(*exp*) stops the execution of the current process until the expression *exp* becomes *true*. The other statements are standard.

```
d0 := 0; d1 := 0; turn := 0;
( while (1) do                               )      ( while (1) do                               )
(   // non-critical section                  )      (   // non-critical section                  )
(   d0 := 1; turn := 0;                      )  ||  (   d1 := 1; turn := 1;                      )
(   wait (d1 = 0 or turn = 1);               )      (   wait (d0 = 0 or turn = 0);               )
(   // critical section                      )      (   // critical section                      )
(   d0 := 0                                  )      (   d1 := 0                                  )
```

Figure 2: Peterson's Algorithm

1.2 The Problem

Figure 2 contains the Peterson's mutual exclusion algorithm for two processes. Let us consider that assignments are atomic operations. Therefore, we are ignoring the presence of *race conditions* (both processes read and write the shared variables, without some kind of explicit synchronization). A naive optimizer could perform the following "optimizations", which would be correct in sequential contexts:

- Propagate "d1 = 0" and "turn = 0" to the statement "**wait** (d1 = 0 **or** turn = 1)", and allowing us to statically evaluate the expression to **true**.
- Propagate "d0 = 0" and "turn = 1" to the statement "**wait** (d0 = 0 **or** turn = 0)", and allowing us to statically evaluate the expression to **true**.

Obviously, such optimizations are incorrect, when we consider the correct (concurrent) semantics of the program. The semantics of the concurrent execution of commands in different processes is modelled by an interleaved execution which proceeds as a sequence of discrete (small) steps. In each step a command is selected in only one of the processes and is executed to completion before the same or another process may initiate an elementary command and proceed to complete it. Since execution of atomic operations is indivisible it is modelled by a single transition.

Therefore, a correct implementation of a data flow analyzer for a parallel language should consider all possible *interleavings* of the small steps of all active processes. It is clear that this may lead to the "state space explosion" phenomenon described in the Model Checking literature [10], due to the exponential number of possible combinations of small steps in a parallel program.

1.3 Related Work

Current approaches in analyzing parallel programs have a restricted model of concurrency. Most of them have a restricted model of shared memory, or even disallow it. Others disallow the dynamic creation of processes, *i.e.* all processes must be statically defined.

Vollmer [18] describes an efficient approach for computing one-bit (also known as bit-vector) data flow analyzers. One-bit data flow analyzers, such as *reaching definitions* and *available expressions*, are based on a two-element semi-lattice. However, such approach does not handle dynamic process creation or dynamic memory allocation, and more complex analysis such as *alias analysis* and *interval analysis* can not be handled.

Srinivasan [15] describes an efficient method of computing the Static Single Assignment Form (SSA) [11] for explicitly parallel programs, however it does not handle dynamic process creation or dynamic memory allocation. Reif [12] investigates the data flow of communicating processes, but these do not have shared variables, and process only communicate through synchronous channels. Cousot [5] extend abstract interpretation to cope with communicating processes, unfortunately his apporach suffers the consequences of the "state space explosion phenomenon", which makes it intractable.

1.4 Contribution of this work

The main contributions of our paper are: a trace based abstract interpretation approach to implement program analyzers for parallel languages; the use of partial order methods from model checking literature [9] to cope with the "state space explosion phenomenon"; the use of constraint solving to implement efficient alias analyzers to allow the use of partial order methods in parallel languages with dynamic memory allocation; an approach to handle parallel languages which does not have the notion of small step (indivisible action); an approach to handle dynamic process creation.

2 Trace Based Abstract Interpretation

Our program analyzers are based on the operational semantics of the language. The operational semantics of a given programming language defines the runtime behavior of programs, and it shall also be called "concrete" semantics or "concrete" interpretation. One way to think about the operational semantics of a language is to regard it as defining an interpreter for such language. We may assume that the operational semantics of a language defines a function $\mathit{successors} : \mathit{State} \to \wp(\mathit{State})$, *i.e.* a function that given a state returns all possible successors of such state. We define *trace* as a finite or infinite sequence of states $(s_0, s_1, \ldots, s_i, \ldots)$, where $s_{i+1} \in \mathit{successors}(s_i)$, *i.e.* there is a transition from s_i to s_{i+1}. Briefly, an interpreter defines all possible *traces* of a given program. The set of *traces* of a particular program is called a *state space* of the program. We shall also use $s_i \to s_j$ meaning $s_j \in \mathit{successors}(s_i)$.

Unfortunately, state spaces of programs are usually infinite or large structures which can not be explicitly represented in computers. Therefore, approximations of language semantics must be used, in which concrete or exact properties are replaced by abstract or approximate properties, e.g. data like 2 and -3 are replaced by *pos* (positive) and *neg* (negative), respectively. The abstract or approximate properties are usually elements of a semi-lattice. The process of "executing" a program using abstract data and/or semantics approximations is called *abstract interpretation* [4]. It is convenient for us to see abstract interpretation as a kind of "symbolic execution".

An abstract interpretation of a program produces an *abstract state space*. Nondeterminism is always present in abstract interpretations where loss of precision makes many language operators nondeterministic. For instance, the statement "if (x > 10) x = x + 1" can not be deterministically executed, if at runtime the value of variable x is *pos*. Nevertheless, if the value is *neg* the body of statement *if* will certainly not be executed.

The concept of *safe simulation* [13] is formalized as a homomorphism from the "*concrete*" state space into the abstract state space. The intuitive rationale for this concept is that any transition in the "concrete" state space is mimicked by one in the abstract state space, *i.e.* any possible behavior of the program is simulated by the abstract interpretation. However, the abstract state space might also contains traces that are not present in the program's concrete state space. The homomorphism from the concrete state space into the abstract state space is defined more formally as:

$$\forall s, s' \cdot s \to s' \Rightarrow \exists a' \cdot \alpha_{state}(s) \to a' \textit{ and } \alpha_{state}(s') \sqsubseteq a'$$

Informally for each concrete transition $s \to s'$ exists an abstract state a', and an abstract transition from the abstraction of s to a' ($\alpha_{state}(s) \to a'$), such that a' is an approximation of s' ($\alpha_{state}(s') \sqsubseteq a'$). Notice that we do not require a' to be the best approximation of s' ($\alpha_{state}(s') = a'$).

Once the abstract state space is built, one can extract analysis information from it. The extracted information is called *collecting semantics*. The "classical" collecting semantics is *state-based*. For example, in *sign analysis* we are often interested in determining the sign of a particular variable in a specific control point (program point). This kind of analysis information can be directly extracted from the abstract state space by using algorithms that traverse it. Abstractly, given an abstract state space t and a control point cp, the analysis information is computed using a function such as:

$$\begin{array}{l} \textit{ana-info} : \textit{Abstract-State-Space} \times \textit{Control-Point} \to \wp(\textit{Abstract-State}) \\ \textit{ana-info}(t, cp) = \{n \mid n \textit{ is a node of } t \textit{ and } \textit{ctrl-point}(n) = cp\} \end{array}$$

where function $\textit{ctrl-point} : \textit{Abstract-State} \to \textit{Control-Point}$ returns the control point associated with a state. The operator $\sqcup$ [1] of the semi-lattice of abstract properties may be used to produce a single abstract state space that approximates the set $\wp(\textit{Abstract-State})$. This way, we might redefine the function as: $\textit{ana-info}(t, cp) = \sqcup\{n \mid n \in t \textit{ and } \textit{ctrl-point}(n) = cp\}$

Several analysis such as reaching-definitions and live-variables [11] are not state-based, because they involve path information. Briefly, the collecting algorithms must compute information about paths in the abstract state space. Abstractly, such algorithm is described as:

$$\begin{array}{l} \textit{ana-info} : \textit{Abstract-State-Space} \times \textit{Control-Point} \to \wp(\textit{Abstract-Paths}) \\ \textit{ana-info}(t, cp) = \{p \mid p \textit{ is a path from } s_0 \textit{ to } s_i \textit{ and } \textit{ctrl-point}(s_i) = cp\} \end{array}$$

The above algorithm reflects the so called forward data flow analysis algorithms such as reaching-definitions [11]. Backward data flow analysis algorithms such as live-variables is defined in a similar way. All path-based analysis are extremely similar to the techniques used in *model checking* of μ-calculus formulas. The model checking algorithms are described in [2], and such relationship was first proposed by Steffen [17] and was also explored by Schmidt [13].

[1] The operator $\sqcup$ is used to "combine" information. For example, $\textit{neg} \sqcup \textit{zero} = \textit{non-pos}$, *i.e.* the "negative" element "combined" with the "zero" element is equal to the "non-positive" element.

To improve efficiency, the two-step process, described above, to implement program analyzers may be combined in a single one. For example, for all state-based analyzers, the abstract state space may be constructed in such way that all states associated with the same program-point are merged, *i.e.* if the program-point associated with *curr-state* is equal to the program-point of an already visited state, then the states are merged using $\sqcup$ operators.

2.1 Partial Order Methods

The approach described above works well for the particular case of sequential languages. Unfortunately, in the case of parallel languages, it suffers the consequences of the "state space explosion phenomenon". Since a parallel program has an exponential number of traces.

We cope with such problem by using partial order methods [9] from the model checking literature. The intuition behind partial order methods is that concurrent executions are really partial orders where concurrent "independent" transitions should be left unordered. Two transitions are independents when the order of their occurrences is incomparable.

It is important to notice that we have not used a direct representation of partial orders. The usual interleaving representation is used, but we attempt to limit the expansion of each partial-order computation to just one of its interleavings, instead of all of them. The efficiency of partial order methods is based on the following assumption:

Assumption 1 *Processes of "real" parallel programs usually perform few interactions.*

Such assumption is reasonable, since it agrees with software engineering principles such as: "program module coupling should be minimized, in the sense of constructing autonomous modules".

Figure 3 contains a very simple parallel program which contains two processes. The marks such as ② are used to represent control points of the program. The first process uses the "local" variable *l1*, and the other uses the "local" variable *l2*. Both of them use the shared variable x. It is clear that accesses to the local variables are independents, thus we may compute just one interleaving in such case. We only need to compute all interleavings when "global" variables are accessed. Figure 4 contains the complete and reduced state spaces associated with the program. Each box represents a state, the label represents the content of the given state. For example, the label "(3,2') l1=1, l2=1, x=$\top$" means that the first process is at control point ③, the second process is at control point 2', and the values of *l1*, *l2* and x are 1, 1, and $\top$ (undefined), respectively.

There are several ways of computing independent transitions which vary in complexity and efficiency. For example, a very simple approach considers that all access to shared variables are dependent, and all other concurrent transitions are independents. A more precise approach will detect if the concurrent processes are only *reading* the value of a shared value, and then classify the respective transitions as independents.

$$
\left(\begin{array}{l}
① \text{ l1 := 0;} \\
② \text{ l1 := l1 + 1;} \\
③ \text{ x := l1;} \\
④
\end{array}\right)
\parallel
\left(\begin{array}{l}
①' \text{ l2 := 1;} \\
②' \text{ l2 := l2 + 2;} \\
③' \text{ x := l2;} \\
④'
\end{array}\right)
$$

Figure 3: Simple Program

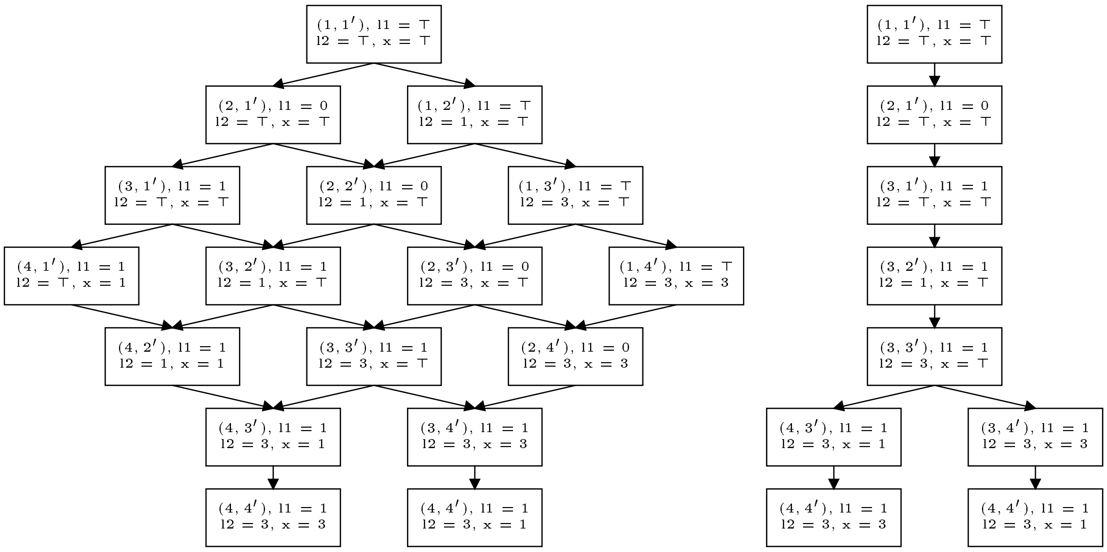

Figure 4: Simple Program Complete and Reduced State Spaces

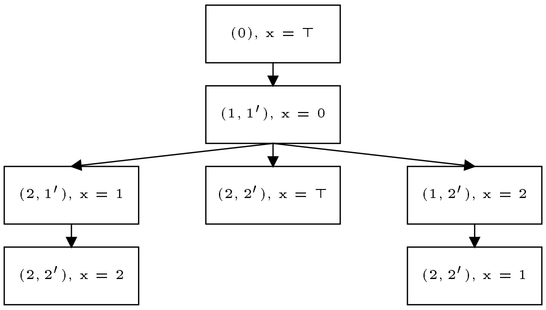

Figure 5: Simulating True Concurrency

2.2 True Concurrency

As stated above, parallelism is modeled by using *interleaving*. A concurrent computation is represented by all possible combinations of "atomic steps", *i.e.* indivisible *actions*. For example, the statement $(x = 0 \parallel x = 1)$ modifies the value of variable x in parallel. By using the interleaved based semantic model, it is clear that the value of variable x is equal to 0 or 1 after the statement execution.

However, we may argue that some programming languages do not have the notion of indivisible *action*. In such languages, even a simple assignment such as $x = 0$ can not be assumed to be executed atomically. The value of variable x after executing a statement such as $(x = 0 \parallel x = 1)$ is not specified, *i.e.* the value may be even different from 0 and 1. Such design decision simplifies the language implementation on hardware platforms, where, for example, the behavior of two different processors updating the same memory cell is completely undefined. In other words, the language compiler does not need to introduce synchronization opcodes for every memory cell access.

Anyway, for our objectives the interleaved based model is sufficient, and the problem related to concurrent assignments is solved by using an extra transition (step). This extra transition represents the simultaneous (or overlapped) update of a given variable, which makes the value of the variable undefined ($\top$). Notice that such transition resembles a synchronization operation. Briefly, the execution of the program

$$0\ \ \mathrm{x} = 0;\ (\ 1\ \ \mathrm{x} = 1\ \ 2\ ) \parallel (\ 1'\ \ \mathrm{x} = 2\ \ 2'\ )$$

produces the trace described in Figure 5. The transition from control point $(1, 1')$ to $(2, 2')$ represents the extra transition described above. It is also wise to associate a "warning message" with the extra transition, since it usually signs incorrect programs.

3 Aliases and Dynamic Memory Allocation

In the parallel imperative language described in previous section, it is extremely simple to determine whether two transitions are independent or not. Basically, it is only necessary to inspect the source code, since the language does not allow us to create aliases. A language supports aliasing if a storage location may be accessed in two or more ways. For example, a C variable may have its address computed and be assigned to or read from both by name and through a pointer. It is necessary to have alias information to determine whether two transitions are independent or not. Let us consider the following program:

x := 0; y := &x; (x := 1 || *y := 2)

The expression $\&x$ returns the address of the variable x, and the statement *y := 2* modifies the storage location pointed by y. Therefore, it is clear that the statements x *:= 1* and *y := 2* are dependent, since both modify the value of variable x.

Alias information is computed by using program analyzers (*alias analyzers*). As described before, our program analyzers reduces the state space by using information related to the dependency between transitions. Thus, we have a recursive problem, since to determine if two transitions are dependent, we need alias information; and to compute alias information, we need information related to transition dependency. However, it is possible to approximate alias information by using a control flow insensitive analyzer [11]. Since such flow analyzer is control flow insensitive, interleaving does not affect it.

We compute alias information by using *set constraints* [8]. Actually, we perform points-to analysis which computes the set of memory locations to which every pointer-valued expression in a program could point. Obviously, alias information may be derived from points-to information.

The approach based on set constraints may be seen as an abstraction of the approaches proposed by Andersen [1] and Steensgaard [16]. The Andersen's points-to analysis is more precise and runs in $O(n^3)$ time, where n is the size of the program. The Steensgaard's analysis is less precise, but runs in almost linear time $O(n\alpha(n, n))$, where α is the inverse of Ackerman's function.

Both Andersen's and Steensgaard's analyses build a *points-to* graph. The nodes of a points-to graph represent memory locations or sets of memory locations, and there must be an edge from x to y if any permutation of the assignment statements in the program may cause location x to contain a pointer to y. Memory locations includes local and global variables, function addresses, string and heap objects.

Our prototype implementation is based on the Andersen's analyses. The result of the flow insensitive alias analysis is used to compute dependent transitions, when we perform more complex analysis implemented by using the approach based on partial order methods. The interested reader should refer [8] to find a detailed explanation of both analyzers.

4 Dynamic Process Creation

Almost all parallel programming languages provides a set of primitive features such as *fork* and *spawn*, which allows us to create an unbound number of processes. To undertake a terminating analysis, some means must be found to limit the dynamically generated processes.

To cope with the problem described above, first we should notice that the program state contains a bag of processes ("process pool"). For example, the initial state of the program

① x = 0 ② || ③ y = 0 ④ || ⑤ z = 0 ⑥

```
① x := 1; ②
while (x = 1) do
    ③ newprocess ( ④ x := 0 ⑤ ); ⑥
⑦
```

Figure 6: Parallel Program with Dynamic Process Creation

may be represented such as "(1,3,5), [x = $\top$, y = $\top$, z = $\top$]", where (1,3,5) is the "process pool", and it asserts the control point where each process is currently in. Therefore, the final state of the above program is "(2,4,6), [x = 0, y = 0, z = 0]".

As stated above, the process pool is a bag of processes, therefore it may be represented by the function of the form *Control-Point* $\rightarrow$ *Nat*. Such function "returns" the number of processes in a particular *control-point*.

Our task is to abstractly represent the process pools, which we do by defining the abstract pools to be a function of the form *Control-Point* $\rightarrow \{0, 1, \infty\}$ [2]. Both domain and codomain of such function space are finite, so there exist a finite number of abstract pools. For an abstract pool *abs-pp* associated with the process pool *pp*, we have that $\textit{abs-pp}(cp) = pp(cp)$, if $cp = 0$ *or* $cp = 1$, and $\textit{abs-pp}(cp) = \infty$, otherwise. Notice that, $\textit{abs-pp}(cp) = \infty$ means that there are 2 or more processes at control point *cp*.

For simplicity, we use the notation $(4, 10^+)$ to represent an abstract pool *abs-pp* such that $\textit{abs-pp}(4) = 1$, $\textit{abs-pp}(10) = \infty$, and $\textit{abs-pp}(cp) = 0$, for every $cp \neq 4$ and $cp \neq 10$. The notation $(\ldots, cp, \ldots)$ [3] represents an abstract pool where $\textit{abs-pp}(cp) = 1$, and $(\ldots, cp^+, \ldots)$ represents an abstract pool where $\textit{abs-pp}(cp) = \infty$. The state of a computation is represented by the tuple $\langle ap, st \rangle$, where *ap* is an abstract pool and *st* is the storage.

By using the above definitions, we may specify the modifications in abstract semantics to handle abstract pools. Briefly, we must specify the new transitions which accounts for abstract pools such as $(\ldots, cp^+, \ldots)$. In such case, we should consider two possible situations:

1. There are exactly two processes at control point *cp*.
   $\langle (\ldots, cp^+, \ldots), st \rangle \rightarrow \langle (\ldots, cp, cp', \ldots), st' \rangle$ if
   $\langle (\ldots, cp, \ldots), st \rangle \rightarrow \langle (\ldots, cp', \ldots), st' \rangle$
2. There are more than two processes at control point *cp*.
   $\langle (\ldots, cp^+, \ldots), st \rangle \rightarrow \langle (\ldots, cp^+, cp', \ldots), st' \rangle$ if
   $\langle (\ldots, cp, \ldots), st \rangle \rightarrow \langle (\ldots, cp', \ldots), st' \rangle$

Figure 6 contains a simple parallel program which dynamically creates processes. The statement "**newprocess**(*stmt*)" creates a new process to execute *stmt*, and continues the execution of the current process. Figure 7 contains the abstract state space for the program described in Figure 6. Let us comment some transitions:

[2] We use the symbol ∞ to represent the case of two or more processes.
[3] We are not using the notation $(\ldots, cp, \ldots)$ to represent infinite pools.

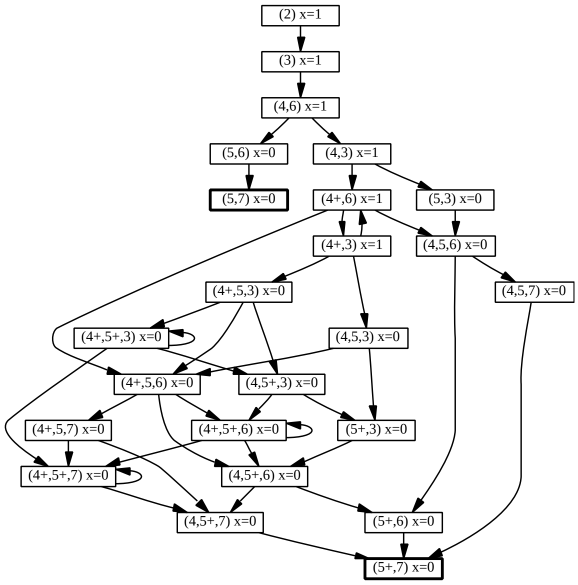

Figure 7: Parallel Program State Space

- $\langle(4,3), x = 1\rangle \rightarrow \langle(4^+,6), x = 1\rangle$, the statement *newprocess* is executed by the process at 3, a new process is created at 4, and process at 3 should continue the execution at 6.

- $\langle(4^+,3), x = 1\rangle \rightarrow \langle(4^+,5,3), x = 0\rangle$, the statement $x := 0$ is executed by one of the processes at 4. This transition assumes that there are more than two processes at 4. On the other hand, the transition $\langle(4^+,3), x = 1\rangle \rightarrow \langle(4,5,3), x = 0\rangle$ assumes that there are exactly two processes at 4.

- The terminal state $\langle(5,7), x = 0\rangle$ represents the trace where the body of the while loop was executed only once. The terminal state $\langle(5^+,7), x = 0\rangle$ represents all traces where loop was executed more than once.

5 Recursive Functions

Recursive functions complicate analysis in general, since multiple instantiations (invocations) of recursive procedures may be active simultaneously. Somehow the analysis must abstract such potentially unbound set of procedure instantiations. Notice that this problem is similar to the one described in previous section (unbound number of processes). Again, the idea is to "join" (merge) related procedure instantiations to ensure termination [6]. However, the use of "join" operations may imply in inadvertently loss of precision. For instance, suppose that we "join" all procedure invocations associated with a procedure f, if f is called from several other procedures g_i, then we will be considering several unrealizable paths.

We minimize such problem defining functions *should-merge* which "decide" whether two procedure instantiations should be merged. Extremely simple functions *should-merge* produce reasonable results. For example, a function *should-*

merge that "joins" two procedure instantiations, if the n most recent call sites are equal. Such function reflects the *nCFA* [14] approach for interprocedural analysis. A more complex function *should-merge* may use information associated with the values of procedure arguments and global variables.

6 Handling "Real" Programs

Global program analysis of "real" (large) programs is usually infeasible even for sequential languages. The cost of such kind of analysis is justified only in re-engineering activities. In the case of sequential languages is usually only performed intraprocedural analysis, *i.e.* the procedures are analyzed independently. Such approach may be also used for parallel languages, however we should always assume that the value of shared variables is $\top$ (undefined), since concurrent processes may modify it. We may improve such approach by using knowledge about language primitives. For example, if the language has primitives to define *critical sections*, then we may assure that the value of shared variables is not modified inside of such sections.

We may also perform the analysis on a module basis, *i.e.* we analyze one (or a group of) program module(s) instead of a single procedure. Therefore, the module private shared variables may be analyzed correctly. A much better analysis may be performed, if we use a model of the environment, where the module will be used. An environment is composed by the rest of the program and the external environment. However, the analysis result will be correct (safe) only if the model correctly reflects (simulates) the environment. Notice that, the input to the program analysis tool will be the module to be analyzed and the model of the execution environment. A similar approach is commonly used in verification activities (model checking) [10].

7 Conclusion

This paper shows how data flow analysis of parallel programs may be implemented by using techniques based on abstract interpretation, model checking and constraint solving. If assumption 1 is valid in a given program, as is usually the case in practice, then the performance of our analyzers for parallel languages is similar to the performance of program analyzers for sequential languages. We have implemented our approach by using the framework described in [6] which was designed to implement language interpreters, program analysis and verification tools.

References

[1] L. Andersen. Program analysis and specialization for the C programming language. Ph.d. thesis, DIKU, University of Cophenhagen, May 1994.

[2] A. Arnold. *Finite Transition Systems.* Prentice-Hall, 1992.

[3] E. M. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[4] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, Jan. 1977.

[5] P. Cousot and R. Cousot. Semantic analysis of communicating sequential processes. In *ICALP'80*, volume 85 of *LNCS*. Springer-Verlag, 1980.

[6] L. M. de Moura, C. J. P. de Lucena, and E. H. Haeusler. Automating the generation of program analysis and verification tools. *submitted to Formal Aspects of Computing*, 2000.

[7] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *Static Analysis, Fourth International Symposium, SAS'97*, 1997.

[8] J. S. Foster, M. Fähndrich, and A. Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical Report UCB/CSD-97-964, University of California at Berkeley, Aug. 1997.

[9] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion*, volume 1032 of *LNCS*. Springer-Verlag, Jan. 1996.

[10] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[11] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[12] J. H. Reif. Data flow analysis of distributed communicating processes. Technical Report TR-12-83, Harvard University, Sept. 1984.

[13] D. A. Schmidt. Trace-based abstract interpretation of operational semantics. *J. Lisp and Symbolic Computation*, 10(3):237–271, 1998.

[14] O. Shivers. Control flow analysis in Scheme. In *SIGPLAN '88 Conference on PLDI, Atlanta, Georgia*, volume 23(7) of *ACM SIGPLAN Not.*, pages 164–174, July 1988.

[15] H. Srinivasan and D. Grunwald. An efficient construction of parallel static single assignment form for structured parallel programs. Technical Report CU-CS-564-91, University of Colorado at Boulder, Dec. 1991.

[16] B. Steensgaard. Points-to analysis in almost linear time. Technical Report MSR-TR-95-08, Microsoft Research, 1995.

[17] B. Steffen. Data flow analysis as model checking. In *Proc. TACS'91*, volume 526 of *LNCS*, 1991.

[18] J. Vollmer. Data flow analysis of parallel programs. Interner Bericht 19/95, Universität Karlsruhe, Fakultät für Informatik, Mar. 1995.