

A New Approach for Concurrent Program Slicing

Pierre Rousseau

CEDRIC - CNAM Paris
292, rue St Martin, 75003 Paris
rousseau@cnam.fr
<http://quasar.cnam.fr>

Abstract. Regarding the progress made in model analysis, more complex models, and consequently more complex programs can now be analyzed. However, this remains a difficult task in particular for concurrent programs which induce a lot of combinatory. Another way to reduce this complexity is to use program decomposition. Program decomposition technics extract a part of a given program while preserving the behavior of the original program w.r.t. a specified property.

QUASAR analyzes concurrent Ada programs, using program slicing as decomposition technic. The program slicer is built using the ASIS tools, that provides syntactic and semantic informations on an Ada source code. These informations can be considered as the “semantic and syntactic graph” mapping an Ada program. This allows to save building the graphs used by traditional program slicing technics and thus to design a simpler and more evolutive algorithm.

This paper presents YASNOST, the program slicer used by QUASAR, describes the method used to slice concurrent Ada programs and illustrates with two significant examples how concurrent programs analysis can take advantage of program slicing for reducing the analyzed program complexity.

1 Introduction

This paper presents a program slicer which doesn't need to build static dependence graphs before slicing a given concurrent program. It records dynamically these dependences when it traverses the syntactic and semantic graph generated by the program compiler. This traversal relies on the standard ASIS tool available for Ada programs. This slicer is part of the QUASAR project developed by our research team.

QUASAR [EKPPR03] is an automatic program analysis tool which aims to formally validate properties of concurrent Ada programs. It generates a formal model from a source code and validates a specified property on the generated model.

The main difficulty of this method is the possible combinatory explosion induced by the process execution interleaving when constructing the reachable state space. To face this problem, QUASAR uses various technics at each step of its analysis process :

1. **program decomposition:** at first, QUASAR uses program slicing in order to reduce the program size while preserving its behavior w.r.t the studied property. The reduced program will help to generate a smaller and simpler state space.
2. **model generation:** this step can be seen as the heart of the QUASAR tool. It translates an Ada program into a corresponding colored Petri Net. The model

construction aims to stay as close as possible of the source code formalism and to produce models trying to limit as much as possible the combinatory of the Ada program [EKPPR03, EKPP⁺05].

3. **model-checking:** at this step, QUASAR uses the model-checker HELENA [Eva05] to verify the property on the colored Petri Net generated at the second step. This tool combines different structural technics and model-checking optimizations in order to deal better with huge state space.

The first step is the most important as it addresses the program at its source. This paper presents YASNOST the Ada program slicer which carries out the program slicing step of QUASAR analysis process.

The concepts and methodology founding QUASAR have been experimented with the Ada language for several concomitant advantages. Ada presents today the most complete and powerful concurrency features. Ada concurrency semantic is well and precisely defined. Ada is currently used for practical and critical applications which need validation. Concurrency analysis methods performed for Ada programs can be used for other languages. For instance, using QUASAR and simulating some Java programs in Ada, we have shown some weakness of Java concurrency semantics [EKPPR06].

2 Program Slicing

Program slicing was first introduced by M. Weiser in [Wei84] and most of the slicing definitions and technics are reviewed in [Tip95] and [XQZ⁺05]. This part of the paper presents the essential definitions used in the whole paper and legitimates the kind of program slicing carried out by YASNOST.

The principle of program slicing is to observe a particular behavior of a program from a specified point of view. The point of view is specified as a *slicing criterion* commonly defined by a couple $\langle n, V \rangle$, with n a statement of the original program and V a set of variables.

The result of the program slicing operation is called a *slice* and can be a set of statements (*non-executable slicing*) or a reduced compilable and executable program (*executable slicing*). This slice must preserve all the behavior of the original program w.r.t. the slicing criterion (ie, at statement n , the values of the variables of the set V have to be computed in the same way in the original program and in the slice). Obtaining a minimal slice is undecidable, however precise slices can be obtained using existing technics.

The slice is obtained by collecting all the parts of a program that may have an effect on the values of a set of variables V at a defined statement n (this is *backward slicing*) or may be affected by the values of the set (this is *forward slicing*).

The kind of program slicing introduced by Weiser is called *static program slicing*. It means that all possible values of program input are considered, thus that all possible executions of the original program have to be considered. The other kind of program slicing is *dynamic program slicing* which considers a particular set of input program values and studies a particular set of executions of the program corresponding to the program execution for these program input values.

YASNOST carries out a static executable backward program slicing. Static because QUASAR validates properties holding for all possible executions of a program; executable for some technical reasons (QUASAR second step process uses computable Ada programs); and backward slicing has been chosen because most of the studied properties validated by QUASAR deal with reachable states of the original program, and thus the slicing criterion definition is closer to the kind of properties analyzed by QUASAR than with forward slicing.

<pre> 1 procedure Robot is 2 3 Nails : Natural := 0; 4 Start : Natural := 0; 5 Total : Natural := 0; 6 Used : Natural := 0; 7 8 begin 9 Get (Start); 10 Total := Nails + Start; 11 Nails := Total; 12 while Nails > 0 loop 13 Nails := Nails - 1; 14 Used := Used + 1; 15 end loop; 16 if Used = 0 then 17 Put ("Nothing done"); 18 end if; 19 Put (Nails); 20 end Robot;</pre>	<pre> 1 procedure Robot is 2 3 Nails : Natural := 0; 4 Start : Natural := 0; 5 Total : Natural := 0; 6 7 8 begin 9 Get (Start); 10 Total := Total + Start; 11 Nails := Total; 12 while Nails > 0 loop 13 Nails := Nails - 1; 14 15 end loop; 16 17 18 19 Put (Nails); 20 end Robot;</pre>
--	--

Fig. 1. Slicing of program Robot with $\langle 19, \{Nails\} \rangle$ as slicing criterion

Figure 1 shows an example of static executable backward program slicing. The program on the left is the original program, and the program on the right is one of its slice observed through the slicing criterion $\langle 19, Nails \rangle$. It means that we want to know the statements which have an effect on the value of the variable `Nails` at line 19. The elements related to the variable `Used` are irrelevant to the value of the variable `Nails` at line 19, thus they do not belong to the slice.

In order to achieve program slicing YASNOST has to be able to define which statements can have an effect on the variables values of the slicing criterion. Weiser had defined two kinds of dependences for sequential program slicing:

- **The control dependence** represents the link between a statement and another statement of which it can control the execution. The most trivial example is the `if-then-else` statement that controls the execution of the statements of its both branches.
- **The data dependence** represents the link between a statement referencing (reading the value of) a variable and the statements defining (modifying) it. For instance, in Figure 1, the statements of line 11 (referencing the variable `Total`) is data dependent on the statement of line 10 (defining `Total`).

This dependence is *transitive*. The modifying statements may reference variables which are also modified in previous statements, and thus these statements have to

be included into the slice because the values of the variables that they modify transitively impact the firstly referenced variable. For instance, as already explained, the statement of line 11 is data dependent on the statement of the line 10. The line 10 references the variable *Start*, defined at line 9. Thus the statement of line 11 is by transitivity data dependent on the statement defining the variable *Start* (line 9).

3 Concurrent Program Slicing

QUASAR analyzes concurrent Ada programs, thus YASNOST has to slice concurrent programs, and to deal with concurrency specific problems described in [Che93, NR00, Kri03], such as dependences introduced by the synchronization between tasks or the non transitivity of the data dependence relation.

To illustrate this last issue considers the following examples (Figure 2).

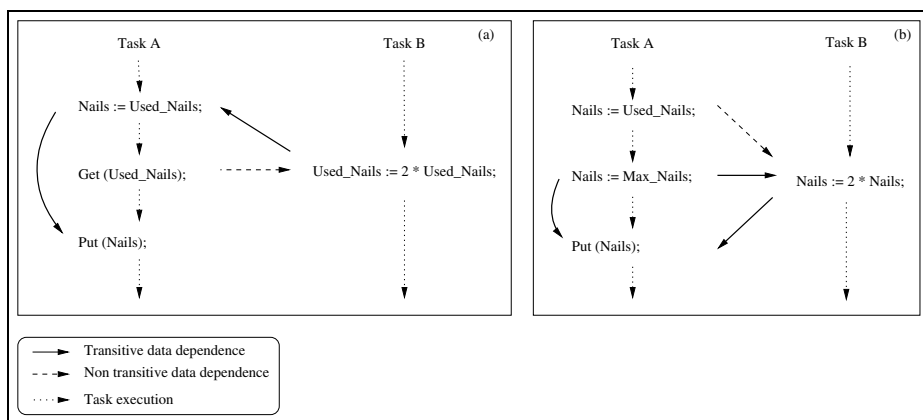


Fig. 2. Examples of imprecise data dependences sequences

In Figure 2 (a), if the data dependence is considered as transitive, the following sequence may be built:

```
{
  Get (Used_Nails);
  Used_Nails := 2 * Used_Nails;
  Nails := Used_Nails;
  Put (Nails);
}
```

Indeed `Put (Nails)` depends on `Nails := Used_Nails` which depends on `Used_Nails := 2 * Used_Nails` which depends on `Get (Used_Nails)`.

However this sequence can't be executed by the program or else it would mean that `Get (Used_Nails)` could be executed before `Nails := Used_Nails` what is impossible.

In the second example (Figure 2 (b)), a variable is modified twice in a task and read and modified in a single statement in another task. In all possible executions, the value of the variable `Nails` is never dependant of the statement `Nails := Used Nails` because the variable `Nails` is always defined by `Nails := Max Nails`. For instance the data dependence relation may take into account this useless sequence:

```
{
  Nails := Used_Nails;
  Nails := Max_Nails;
  Nails := 2 * Nails;
  Put (Nails);
}
```

In both cases, considering the dependence relation as transitive leads to take into account sequences of statements that are impossible (first case) or useless (second case). So the resulting slice will contain statements that do not affect the slicing criterion and thus is imprecise.

Previous works [CX01, Kri03] are all based on an augmentation of the dependence graph approach. These graphs are complex and contain all possible dependences relation between all the program statements. YASNOST, the QUASAR slicer, relies on another concurrent program slicing approach which is based on ASIS, an Ada tool which allows to inspect the syntactic tree of an Ada program by using the semantic links existing between its elements. Instead of building a static dependence graph, YASNOST records dependences “on the fly” which naturally avoid to build useless dependences.

4 YASNOST

At the moment, YASNOST supports the basic Ada language (assignment, conditioned statements, ...), subprograms and the part of the language related to concurrency (tasks, protected object, rendez-vous, ...). Pointers and dynamic structures are not yet supported except the dynamic task allocation. Unstructured control flow such as `exit` statements are supported but exceptions and jumps are not. However, non supported parts of the language can be wholly included into the slice.

4.1 Tree Manipulator: ASIS

ASIS (Ada Semantic Interfaces Specification [ISO95]) is an interface between an Ada environment and tools requiring static information about the syntax and the semantic of an Ada program.

ASIS provides two ways for obtaining information about an Ada source code. First, there is an iterator allowing traversing the syntactic tree of an Ada program with a depth-first left-hand method. The second tool is a set of queries that allows the user to navigate in the syntactic tree following semantical dependences between its nodes. The tree associated with the ASIS queries can be view as a “syntactic and semantic” graph.

Figure 3 shows an example of ASIS graph used to get information about a source code. The plain lines represent the syntactic tree of the Ada program of Figure 1 (original program on the left). The dashed line represents an ASIS query linking an identifier

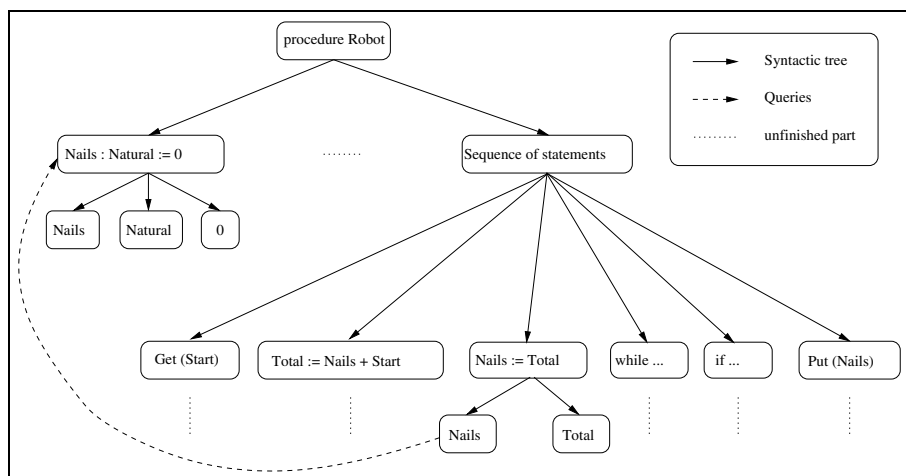


Fig. 3. Part of an ASIS graph of program of Figure 1

(Nails) to its declaration (Nails : Natural := 0). Note that the ASIS syntactic tree has been simplified for the sake of simplicity.

The first way to use this interface is to implement already known program slicing algorithms, as done in [SC03]. A second way is to deduce the dependence graph from ASIS syntactic tree. YASNOST uses the ASIS tools in a third way : build dependences “on the fly” with the ASIS queries.

4.2 Algorithm

The algorithm used by YASNOST aims to separate as much as possible the search of the different kind of dependences. YASNOST uses a stack in which are pushed the elements (nodes of the syntactic tree) of which YASNOST needs to check the dependences and thus that have to be kept. The following algorithm is carried out:

1. Push statement of the slicing criterion into the stack. With YASNOST, the user can use annotations to specify some statements to keep in the slice. These statements are also pushed into the stack at this step.
2. Push instantiated tasks into the stack. At this step, each task declaration and allocation is kept. This step is done traversing the syntactic tree and collecting every task declarations and every statements allocating dynamically a task.
3. Looking for dependences. This step is repeated as long as the stack is not empty
 - (a) Pop the first element of the stack and add it to the list of the kept nodes.
 - (b) Push control dependent statements into the stack.
 - (c) Push data dependent statements into the stack
 - (d) Push all declarations related to the popped element into the stack. This step is carried out in order to have an executable slice.
4. Create the executable slice with the list of kept nodes.

The parts related to control dependences and the data dependences, in particular when data dependences have to be checked through parameters passing, have to be detailed.

4.3 Control Dependences

In order to collect the control dependences YASNOST uses the weak and strong dependences defined by Cheng [Che93] as follow:

- Strong control dependence is the dependence explained in section 2. ASIS provides a query giving the father node of any node of the syntactic tree. Every father is pushed into the stack. By this way all elements enclosing the popped element are pushed into the stack.
- Weak control dependence corresponds to statements that depend on the termination of another statement (for instance a statement following a loop). If the popped element is a body, YASNOST pushes into the stack all statements related to concurrency and all statements that may not terminate or that may terminate another statement:
 - loops the termination of which cannot be statically evaluated.
 - the calls to protected sub-program (in Ada protected objects are Hoare monitor constructs [Hoa74] with function, procedure and guarded entries).
 - rendez-vous statements.
 - some unstructured control flow statement are also kept at this step. For instance, `exit` statements included in a kept loop. Jump statements such `goto` are not yet supported by YASNOST.

4.4 Data Dependences

In order to find data dependences, YASNOST follows the algorithm described Figure 4. This algorithm uses two lists:

- `Read` : a list of variables for which YASNOST searches statements modifying them.
- `Writers` : the list of elements modifying at least a variable from `Read`. ASIS provides a set of queries allowing to retrieve all statements that contains an identifier corresponding to a given declaration. Statements which belong to the same task are sorted by their textual position (line and column numbers).

YASNOST also uses a transitivity graph for all the variables of the slicing criterion. These graphs are built using sequences of transitive data dependences between the program statements. These graphs are built on the fly. If a data dependence between two statements would lead to only build paths in the transitivity graph such the ones described in section 3, the data dependence is not transitive and thus the statement from the `Writers` set is not added to the slice.

In order to realize this, YASNOST has to be able to know when a statement can be executed before another. The precedence between statements which belong to the same task is checked as follow:

- if two statements are in the same body, the precedence is determined by the textual position (line and column number). If both statements belong to the same loop, they are considered as mutual predecessors.
- if they don't belong to the same body, it means that they are in different subprograms, then, the precedence between the calls (and between both statements) are checked.

If the statement `Writer` has to be added to the slice, then the modified variable is removed from the `Read` set and from the slicing criterion set. Thus for any other elements of the set `Writers`, YASNOST has to check again if it modifies an element which still belongs to the `Read` set. This is done to avoid to include in the slice old and useless variable value modification.

Statements which belong to different tasks are considered as mutual predecessors. In this case, the transitivity only is checked.

```

Read ← Read_By (Current_Element) & Variables (Slicing_Criterion);
Writers ← Modifying (Read)
for reverse Writer of Writers loop
  if Same_Task (Writer, Current_Element) then – transitivity holds
    if Modify (Writer, Read) and
      Precedence (Writer, Current_Element) – check precedence
    then
      Push (Writer, Stack);
      Remove (Modified_Variables (Writer), Read);
      Remove (Modified_Variables (Writer), Slicing_Criterion);
      Add (Writer, Current_Element, Transitivity_List);
    end if;
  else – precedence holds
    if Transitive (Writer, Transitivity_List) then – check transitivity
      Push (Writer, Stack);
      Add (Current_Element, Transitivity_List);
    end if;
  end if;
end loop;

```

Fig. 4. Part of algorithm to find data dependences

4.5 Inter-procedural Slicing Issues

When slicing an inter-procedural program, YASNOST has to deal with parameter passing, and to retrieve the parameters of which the final values computed by the subprograms (the parameters modifications or the return value of the sub-program) depend.

As YASNOST produces executable slices, all calls to a procedure have to be written with all effective parameters corresponding to all formal parameters used by the declaration of the called sub-program.

This could lead to build imprecise slice. For instance, consider the procedure `PROC` of Figure 5, which is sliced in order to know the value of the variable `Arg4` at line 28. All the parameters of the procedure `PROC` have to be added to the slice. But only two of them are useful at the call at line 21; the others are also in the slice because of the call at line 26. Thus, at the procedure call at line 21, without information linking the modified parameters to the parameters used to modify them, lines 19 and 20 would be added to the slice as `Arg1` and `Arg2` are referenced by this last call. This would be

imprecise since line 21 is in the slice because this statement is needed to evaluate the condition of the if statement at line 23 which uses only the values of `Arg1` and `Arg3`. The value of `Arg4` is newly defined at line 22 so the value of `Arg4` computed at line 21 (the call) is not relevant to the final value of `Arg4`. And then `Arg1` which is used to compute the value of `Arg4` through the call should not be considered and the statement defining it at line 19 should not belong to the slice.

<pre> 1 procedure Example is 2 procedure Proc 3 (In1 : in Integer; 4 In2 : in Integer; 5 Out1 : out Integer; 6 Out2 : out Integer) 7 is 8 begin 9 Out1 := In1; 10 Out2 := In2; 11 end Proc; 12 13 Arg1 : Integer := 0; 14 Arg2 : Integer := 0; 15 Arg3 : Integer := 0; 16 Arg4 : Integer := 0; 17 18 begin 19 Arg1 := 1; 20 Arg2 := 2; 21 Proc (Arg1, Arg2, Arg3, Arg4); 22 Arg4 := 4; 23 if Arg3 > Arg1 then 24 Arg2 := 6; 25 Arg1 := 7; 26 Proc (Arg1, Arg2, Arg3, Arg4); 27 end if; 28 Put (Arg4); 29 end;</pre>	<pre> 1 procedure Example is 2 procedure Proc 3 (In1 : in Integer; 4 In2 : in Integer; 5 Out1 : out Integer; 6 Out2 : out Integer) 7 is 8 begin 9 Out1 := In1; 10 Out2 := In2; 11 end Proc; 12 13 Arg1 : Integer := 0; 14 Arg2 : Integer := 0; 15 Arg3 : Integer := 0; 16 Arg4 : Integer := 0; 17 18 begin 19 Arg1 := 1; 20 21 Proc (Arg1, Arg2, Arg3, Arg4); 22 Arg4 := 4; 23 if Arg3 > Arg1 then 24 Arg2 := 6; 25 26 Proc (Arg1, Arg2, Arg3, Arg4); 27 end if; 28 Put (Arg4); 29 end;</pre>
--	--

Fig. 5. Example of inter-procedural slicing

In [HRB90], to slice sequential inter-procedural programs, the authors use summary arcs in their dependence graph in order to know for every call which parameter have an effect on the results produced by the sub-program. But, as pointed Krinke, due to the non-transitivity of the data dependence, these arcs can't be used to slice concurrent programs using dependence graphs. It is shown Figure 6. If all the dependences were considered transitive then the statement `UsedNails := UsedNails + 1` would have to be included into the slice because `Nails := UnusedNails` is considered as transitively dependent of statement `UnusedNails := TotalNails - UsedNails`. Then the summary arc would link the variable `Nails` with the variable `UsedNails` and thus, the call to `UpdateNails (UsedNails, Nails)` would be considered as referencing the `UsedNails` variable. This would build an imprecise slice, since in concurrent programs, data dependence is not transitive and thus the value of the parameter `UsedNails` has no effect on the final value of `Nails`.

However as YASNOST builds only transitive data dependency sequences, and thus builds a transitivity graph where there is always at least one path from a statement to

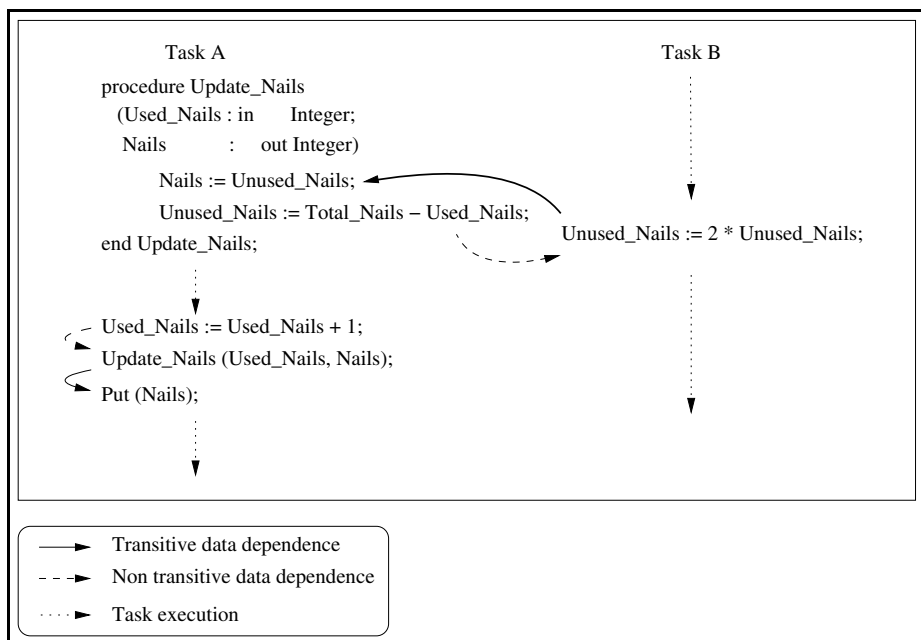


Fig. 6. Example of precise slice for inter-procedural concurrent program

another one which belongs to a transitive data dependence sequence of statements, it can build these summary arcs also in a concurrent context. As shown Figure 6, only the plain arrows are considered as transitive data dependencies, the dashed ones are not considered and thus are not added to the slice.

5 Examples

5.1 The Robot Example

The example presented Figure 7 is a simple robot which plants nails. This program uses three tasks : the main program, the task managing the right arm and the task managing the left arm. It shows how program slicing reduces a program size and thus helps to debug it.

The main program starts by asking how many times the nails box should be filled when empty, then just surveys that the nail box always contains at least one nail. If not, it calls the right arm to fill the mailbox. When the max number of filling has been reached, the program stops. The left arm places the nail to plant (if there is at least one nail in the box) and then asks the right arm to hit the nail with the hammer. The right arm waits orders and either fills the nail box or hits a nail with a hammer when asked.

Even if this source code is simple, it may be difficult to understand its behavior and then to find bugs.

Suppose that one wants to check properties related to the value of `Nails` at the line 90 (the last statement of the main procedure). So the slice will be obtained by slicing the original program with the slicing criterion $\langle 90, \{Nails\} \rangle$, and will allow to focus on the statements that may have an effect on the variable `Nails`. The sliced program is more clear. Before the model-checking step one bug can already be discovered when looking at the use of the variable `Nails` : at line 71, when it fills the box, right arm removes 10 nails from the count instead of adding them to the count. Afterwards model-checking can be used to find more subtle mistakes, or to formally prove some property about the program variables taking advantage of the reduced size of the slice.

Table 1. Part of the report generated by YASNOST after slicing the program of the Figure 7

sliced statements	::	16 (55%)
sliced functions	::	-
sliced procedures	::	6 (86%)
sliced entries	::	0 (0%)
sliced variables	::	2 (33%)

Table 1 displays some results about the slicing operation of the robot original program. Although the result of slicing largely depends on the way the program has been written, this table shows that more than 50% of the program statements have been sliced. Assuming that the sliced procedures could be much more complex than a simple output, the sliced statements ratio could be widely larger without increasing the computation time which is instantaneous to slice this program.

5.2 The Client-Server Example

The second example (Figure 8) shows a more subtle possible use of slicing operation. YASNOST slices concurrent programs in order to make easier the model-checking step of QUASAR. The size of the generated model and, in most cases, the size of the state space, are intuitively related to the size of the studied program.

But for concurrent programs, the size of the state space is more related to the combinatory induced by the indeterminism of concurrency than to the size of the program. Our second example shows that even when the original program is not significantly reduced, the slicing operation may be useful by removing a lot of combinatory from the original program.

This example (Figure $\langle 64, \emptyset \rangle$) is a simple client-server architecture where the server dynamically allocates a task for every client accessing its services. Here presence of deadlock is checked ; thus the slicing criterion $\langle 64, \emptyset \rangle$ is used (last statement, no variables).

As recorded Table 2, the reduction obtained by program slicing operation is small, but, when checking presence of deadlocks, QUASAR use shows that the two lines removed from the original program were generating a lot of combinatory that led to the state space explosion.

Table 2. Part of the report generated by YASNOST after slicing the program of the Figure 8

sliced statements	:: 2 (18%)
sliced functions	:: -
sliced procedures	:: 0 (0%)
sliced entries	:: 0 (0%)
sliced variables	:: 1 (17%)

Here the slicing operation didn't remove a lot of statements (although the statements computed by the procedure `Get_Value` could be much more complicated as it is supposed to compute the requested service offered by the server) but removed a lot of complexity of the program as shown in Table 3 in a instantaneous time which as to be compared to the time needed to compute the nodes of the state space (which is naturally long).

Table 3. State space generated by HELENA for the model of the program of Figure 8

Clients	Running tasks	Reachable states	Reachable states with slicing
1	4	247	221
2	6	9 499	5 939
3	8	735 767	239 723
4	10	-	12 847 017

6 Related Works

A slicer for Ada programs already exists, Adaslicer [SC03], but it operates only on sequential programs. Few other tools have been designed for slicing concurrent Java and C ANSI programs [DCH⁺99, Kri03, Zha99]. Only [Kri03] builds slices which take into account the non-transitivity of data dependence in a concurrent context and demonstrates that the slices are more precise and more quickly computed.

These approaches use augmentation of the dependence graphs and build all the dependences between all the program statements. Thus they will have better execution times than YASNOST when computing a lot of slices for a given program, but will be slower for a unique slice since they have to build the complete dependence graphs while YASNOST records dependences only w.r.t. the statements which already belong to the slice. As YASNOST is the first step of QUASAR which carries out a formal analysis by model-checking, building all possible slices of a program is not necessary.

7 Conclusions and Further Works

This paper has shown how static analysis can greatly help formal analysis to deal with large and complex programs by removing useless statements regarding the property to check and also by removing a lot of complexity from these programs. Other static

```

1 with Ada.Text_IO;           use Ada.Text_IO;
2 with Ada.Integer_Text_IO;  use Ada.Integer_Text_IO;
3
4 procedure Robot is
5
6   Used_Nails : Natural := 0;
7   Nails      : Natural := 0;
8
9   task type Left_Arms;
10  Left_Arm : Left_Arms;
11
12  task type Right_Arms is
13  entry Hit_Nail;
14  entry Fill_Nails_Box;
15  end Right_Arms;
16  Right_Arm : Right_Arms;
17
18  task body Left_Arms is
19  procedure Take_Nail is
20  begin
21    Put_Line ("Left arm took a nail");
22  end Take_Nail;
23
24  procedure Set_Nail is
25  begin
26    Put_Line ("Left arm set the nail");
27  end Set_Nail;
28  begin
29  loop
30    if Nails > 0 then
31      Take_Nail;
32      Nails := Nails - 1;
33      Set_Nail;
34      Right_Arm.Hit_Nail;
35    end if;
36  end loop;
37  end Left_Arms;
38
39  task body Right_Arms is
40  procedure Take_Hammer is
41  begin
42    Put_Line ("Right arm took the hammer");
43  end Take_Hammer;
44
45  procedure Hit_Nail_3_Times is
46  begin
47    Put_Line ("Right arm hit to the nail");
48  end Hit_Nail_3_Times;
49
50  procedure Taking_Nails_From_Reserve is
51  begin
52    Put_Line ("Right arm took nails from the reserve");
53  end Taking_Nails_From_Reserve;
54
55  procedure Put_Nails_In_Box is
56  begin
57    Put_Line ("Right arm fill the nails box");
58  end Put_Nails_In_Box;
59  begin
60  loop
61    select
62      accept Hit_Nail do
63        Take_Hammer;
64        Hit_Nail_3_Times;
65        Used_Nails := Used_Nails + 1;
66      end Hit_Nail;
67    or
68      accept Fill_Nails_Box do
69        Taking_Nails_From_Reserve;
70        Put_Nails_In_Box;
71        Nails := Nails + 10;
72      end Fill_Nails_Box;
73    end select;
74  end loop;
75  end Right_Arms;
76
77  Filling : Natural := 0;
78  U_Check : Natural := 0;
79
80  begin
81  Get (Filling);
82  while Filling > 0 loop
83    if Nails < 1 then
84      Right_Arm.Fill_Nails_Box;
85      Filling := Filling - 1;
86    else
87      U_Check := U_Check + 1;
88    end if;
89  end loop;
90  Put (Nails);
91  end Robot;

```

```

1
2
3
4 procedure Robot is
5
6   Nails      : Natural:=0;
7
8   task type Left_Arms;
9   Left_Arm : Left_Arms;
10
11  task type Right_Arms is
12  entry Hit_Nail;
13  entry Fill_Nails_Box;
14  end;
15  Right_Arm : Right_Arms;
16
17  task body Left_Arms is
18
19
20
21
22
23
24
25
26
27
28  begin
29  loop
30    if Nails > 0 then
31
32      Nails := Nails - 1;
33
34      Right_Arm.Hit_Nail;
35    end if;
36  end loop;
37  end;
38
39  task body Right_Arms is
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59  begin
60  loop
61    select
62      accept Hit_Nail do
63
64
65        null;
66      end;
67    or
68      accept Fill_Nails_Box do
69
70
71        Nails := Nails + 10;
72      end;
73    end select;
74  end loop;
75  end;
76
77  Filling : Natural:=0;
78
79
80  begin
81  while Filling > 0 loop
82    if Nails < 1 then
83      Right_Arm.Fill_Nails_Box;
84      Filling := Filling - 1;
85    end if;
86  end loop;
87
88
89  end;
90
91  end;

```

Fig. 7. Slicing of Robot program with $\langle 90, \{Nails\} \rangle$ as slicing criterion

analysis approach could be used such as settling variable limits in order to limit at the very most the size of types used and thus help to reduce the size of the state space.

The slicing algorithm carried out by YASNOST is more adapted than previous ones to be a first step of a complete formal analysis process, such the one performed by QUASAR, saving time and resources for the long and complex model-checking step

```

1 procedure Server is
2
3   Max_Client : Integer := 5;
4
5   protected type Datas is
6     procedure Get_Value (Value : out Integer);
7   private
8     Data_Value : Integer := 0;
9   end Datas;
10
11  protected body Datas is
12    procedure Get_Value (Value : out Integer) is
13      begin
14        Data_Value := Data_Value + 1;
15        Value := Data_Value;
16      end Get_Value;
17  end Datas;
18
19  Data : Datas;
20
21  task type Thread is
22    entry Get_Value (Param : out Integer);
23  end Thread;
24  type Access_Thread is access Thread;
25
26  task body Thread is
27  begin
28    accept Get_Value (Param : out Integer) do
29      Data.Get_Value (Param);
30    end Get_Value;
31  end Thread;
32
33  task type Task_Server is
34    entry Get_Thread (Id : out Access_Thread);
35  end Task_Server;
36
37  task body Task_Server is
38  begin
39    for I in 1..Max_Client loop
40      accept Get_Thread (Id : out Access_Thread) do
41        Id := new Thread;
42      end Get_Thread;
43    end loop;
44  end Task_Server;
45
46  The_Task_Server : Task_Server;
47
48  task type Client;
49  type Access_Client is access Client;
50
51  task body Client is
52    Id : Access_Thread;
53    Value : Integer;
54  begin
55    The_Task_Server.Get_Thread (Id);
56    Id.Get_Value (Value);
57  end Client;
58
59  A_Client : Access_Client;
60 begin
61  for I in 1..Max_Client loop
62    A_Client := new Client;
63  end loop;
64 end Server;

```

```

1 procedure Server is
2
3   Max_Client : Integer:=5;
4
5   protected type Datas is
6     procedure Get_Value
7   private
8
9   end;
10
11  protected body Datas is
12    procedure Get_Value
13      begin
14
15        null;
16      end;
17  end;
18
19  Data : Datas;
20
21  task type Thread is
22    entry Get_Value
23  end;
24  type Access_Thread is access Thread;
25
26  task body Thread is
27  begin
28    accept Get_Value
29      Data.Get_Value
30    end;
31  end;
32
33  task type Task_Server is
34    entry Get_Thread (Id : out Access_Thread);
35  end;
36
37  task body Task_Server is
38  begin
39    for I in 1..Max_Client loop
40      accept Get_Thread (Id : out Access_Thread) do
41        Id := new Thread;
42      end;
43    end loop;
44  end;
45
46  The_Task_Server : Task_Server;
47
48  task type Client;
49  type Access_Client is access Client;
50
51  task body Client is
52    Id : Access_Thread;
53  begin
54    The_Task_Server.Get_Thread (Id);
55    Id.Get_Value
56  end;
57
58
59  A_Client : Access_Client;
60 begin
61  for I in 1..Max_Client loop
62    A_Client := new Client;
63  end loop;
64 end;

```

Fig. 8. Client-Server example $\langle 64, \emptyset \rangle$ as slicing criterion

which follows the slicing step. Programs written in other programming languages could be sliced using the technics presented in this paper, however the semantic and syntactic information on the tree representation of the program as provided by ASIS (such as the query linking the node of an identifier and the node of its declaration) has to be obtained.

References

- [Che93] Jingde Cheng. Slicing concurrent programs - a graph-theoretical approach. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 223–240. Springer-Verlag, 1993.
- [CX01] Zhenqiang Chen and Baowen Xu. Slicing concurrent java programs. *SIGPLAN Not.*, 36(4):41–47, 2001.

- [DCH⁺99] Matthew B. Dwyer, James C. Corbett, John Hatcliff, Stefan Sokolowski, and Hongjun Zheng. Slicing multi-threaded java programs: A case study. Technical Report 99-7, KSU, 1999.
- [EKPP⁺05] Sami Evangelista, Claude Kaiser, Jean François Pradat-Peyre, Christophe Pajault, and Pierre Rousseau. Dynamic tasks verification with QUASAR. In *International Conference on Reliable Software Technologies (Ada-Europe)*, volume 3555, page 91. Springer-Verlag, June 2005.
- [EKPPR03] Sami Evangelista, Claude Kaiser, Jean François Pradat-Peyre, and Pierre Rousseau. Quasar, a new tool for concurrent ada program analysis. In *International Conference on Reliable Software Technologies (Ada-Europe)*, volume 2655, pages 168–181. Springer-Verlag, June 2003.
- [EKPPR06] Sami Evangelista, Claude Kaiser, Jean François Pradat-Peyre, and Pierre Rousseau. Comparing Java, C# and Ada monitors queuing policies : a case study and its ada refinement. In *Ada Letters*. ACM Press, 2006.
- [Eva05] Sami Evangelista. High level petri nets analysis with helena. In *26th International Conference on Applications and Theory of Petri Nets 2005, ICATPN 2005*, volume 3536, page 455. Springer-Verlag, 2005.
- [Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.
- [HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.
- [ISO95] ISO/IEC-15291. Ada semantic interface specification. 1995.
- [Kri03] Jens Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 178–187. ACM Press, 2003.
- [NR00] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 180–190. ACM Press, 2000.
- [SC03] Ricky E. Sward and A.T. Chamillard. Adaslicer: an ada program slicer. In *Proceedings of the 2003 annual international conference on Ada*, pages 10–16. ACM Press, 2003.
- [Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
- [Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
- [XQZ⁺05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.
- [Zha99] Jianjun Zhao. Slicing concurrent java programs. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, page 126. IEEE Computer Society, 1999.