Slicing Promela and its Applications to Model Checking, Simulation, and Protocol Understanding

Lynette I. Millett and Tim Teitelbaum[*]
(millett@cs.cornell.edu, tt@cs.cornell.edu)
Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

Static program slicing has been used effectively for a variety of applications ranging from debugging to program integration to software re-engineering. A program slice consists of the parts of a program that may affect or are affected by the value being computed at the point of interest. A slice, for sequential programs, is computed by examining control and data dependence in the program. Recent work in slicing concurrent programs examines how values at a particular program point are affected by synchronization, communication, and non-determinism (along with the traditional control and data dependence effects.) We are extending this work to slice the Promela programming language, used to specify protocols for the Spin model checker. Another application of slicing may be its usefulness in paring down protocol descriptions to just the pieces that affect particular points of interest (e.g. assertion statements, never claims, etc. in Promela). Model checking and simulation of the pared-down protocol may, in some cases, be much more efficient. We present program slicing as a tool that, along with model checking and simulation techniques, can facilitate understanding and debugging of protocols.

Introduction

A static program slice consists of the parts of a program that affect (backward slice) or are affected by (forward slice) a set of points in the program. Slicing[1] has been used effectively for a broad range of applications including debugging, program differencing, re-engineering, and program understanding. Thus far, however, there has been little work done on slicing concurrent programs. Cheng and Krinke have addressed this issue [4, 12]. Cheng slices an Occam-like language without dynamic process creation. Krinke has developed an algorithm for slicing in the presence of interleaved code. In this paper we extend Cheng's work to slicing Promela.

Our initial subject of inquiry was slicing of concurrent languages. Promela has several features that make it a good candidate for such an investigation: channels as first-class objects, both synchronous and asynchronous communication, dynamic process creation, and, more practically, its similarity to C, slicing of which is well-understood [17]. We describe how we will handle the distinctive aspects of Promela in the PDG/SDG representation employed by Grammatech's program slicer [8], which is based on the Wisconsin Program Slicing tool [17]. As Krinke points out, Cheng's approach (and therefore ours) has limitations, most notably with respect to precision. However, we show that it will be possible to get useful results even with somewhat imprecise Promela slices.

Although we were originally drawn to Promela as a convenient vehicle for studying how to slice concurrent features of programming languages, Clarke subsequently suggested to us [7] that such work has potential for significantly speeding up simulation and model checking, and thus might have considerable practical consequence. In this paper, we present some experimental evidence for this hypothesis.

---

[*] The authors gratefully acknowledge the support of the Office of Naval Research under contract No. N00014-92-J-1973.

[1] For the rest of the paper, we use the terms 'slicing' and 'static slicing' interchangeably. There is a notion of dynamic slicing (see [15] for an overview), but here we are concerned exclusively with static slicing.

Slicing Promela Protocols – Three Motivating Examples

Before we discuss the details of slicing, and, in particular, what slicing a language like Promela entails, we present concocted examples of protocols that demonstrate the promise of slicing for speeding up simulation and model checking. Our claim is that slicing is useful in cases where there are relatively independent pieces of a protocol that are of interest to the user. That is, perhaps a user is interested in examining a particular property, assertion, or even variable, in the protocol in isolation. If so, slicing provides a way to focus the inquiry on just those parts of the protocol affecting the point(s) of interest. Here, we provide simple examples of such cases. It should be clear that these examples are generalizable to larger, more interesting protocols.

Consider the following simple protocol in which processes A and B are completely independent:

```
int y = 1;
int x = 1;

proctype A()
{
      do
      :: (x > 0) -> x--;      printf("x = %d\n", x);
      :: (x < 0) -> x++;      printf("x = %d\n", x);
      :: (x == 0) -> break;
      od;
}
proctype B()
{
      do
      :: (y >= 0) -> y++;     printf("y = %d\n", y);
      :: (y < 0) -> y--;      printf("y = %d\n", y);
      :: (y == 0) -> break;
      od
}
init
{
      run B();
      run A();
}
```

In this example, it is clear that a simulation will not terminate due to the infinitely increasing value of `y` in process B. However, were a user interested in running and examining the output of a simulation of a modified protocol consisting of just what involves the value of `x`, then slicing provides a way to generate such a modified protocol. A forward slice with respect to the declaration of `x` would include all definitions and uses of `x` and anything else that depends on them. In the above case, such a slice would include at least the protocol statements that are boldfaced and would not include any of the statements in process B (which cause nontermination.) In general, slicing with respect to points of interest in a program and then running Spin on the sliced version can reduce both the amount of time required for a simulation and the amount of output generated by a simulation. The former is of definite practical interest, and the latter is useful in understanding output produced by the Spin simulator.

As an example of a protocol for which slicing provides dramatic effects on the amount of resources required for model checking, consider the following:

```
mtype = { callA, callB } ;
chan Reqs = [1] of { int } ;
proctype manager()
{
      int count1 = 0;
      int count2 = 0;
      do
      :: Reqs?callA -> run A(); count1++;
      :: Reqs?callB -> run B(); count2++;
      :: (count1 + count2 >= MAX_REQS) -> break;
      od;
      assert(count1 + count2 >= MAX_REQS);

}
proctype requestor()
{
      int count = 0;
      do
      :: (count <= MAX_REQS) -> Reqs!callA; count++;
      :: (count <= MAX_REQS) -> Reqs!callB; count++;
      :: (count >= MAX_REQS) -> break;
      od
}
proctype A()
{
/* code that recursively spawns many more A processes as well as B
processes up until the number of times A has been spawned reaches a
certain limit */
}
proctype B()
{
/* increment a global variable, print a diagnostic */
}
init
{
      run manager();
      run requestor();
}
```

Process *manager* takes requests from the process *requestor* and, depending on their type, starts process A or process B. Process A either calls itself, spawning more A processes, or calls B. Termination occurs when all instantiated versions of process A 'realize' that a global variable is greater than some constant. (We've omitted the details of this to save space.) Suppose we wish to slice with respect to the assertion in process *manager* that states that the total number of requests is at least `MAX_REQS`. (This assertion is boldfaced and enlarged in the source code above.) A backward slice with respect to this assertion would include (at most) the code that is boldfaced.

When performing a full search on the unsliced version, where `MAX_REQS` is set to 3, Spin Version 3.0.5 runs out of memory after generating/searching more than 2 million states. For the sliced version, Spin finishes promptly after having to examine only 305 states. Note that validity of assertions does have an impact. In this example, if the assertion is false, Spin catches it very quickly in both cases (although, of course, more quickly in the sliced version). For instance, if we change the assertion to `assert(count1 +`

`count2 == MAX_REQS`), Spin finds the assertion violation after 162 states in the unsliced version. In the sliced version it requires just 30 states.

We have described two types of problems for which slicing can be helpful. One was demonstrated by a protocol in which the simulation would not terminate and the other by a case in which model checking to verify an assertion statement was computationally infeasible. Slicing can also be used with respect to properties ("never claims" in Promela.)

To slice with respect to a never claim is simply to slice with respect to each "statement" in the never claim. In the program representation, a never claim is treated as a separate "process" (processes, for the moment, are treated analagously to procedures in the PDG/SDG representation we are extending. See [10, 11] for discussions of interprocedural slicing.) However, "never claims" in Promela are not allowed side effects. In this respect, they can be thought of as only manipulating global variables/channels. The dependence edges into the never claim are thus from the declarations of these variables and channels to their uses within the never claim. Thus, our (admittedly imprecise) slice with respect to a never claim conceptually involves a backward step[2] along these edges to the declaration of each variable in the never claim, followed by a forward slice from those declarations. In this way, any use in the protocol of a variable mentioned in the never claim will end up in the slice.

Consider the following never claim, which could be added to the above protocol. This never claim expresses the following behavior: The difference between the variables `count1` and `count2` should not remain large (in this case greater than 10) forever.

```
never {
      do
      :: skip
      :: (count1 - count2 > 10) -> break
      od;
accept: do
      :: (count1 - count2 > 10)
      od
}
```

Once again, Spin's performance on the sliced and unsliced versions is significantly different. When we slice with respect to this never claim, since the only variables of interest are `count1` and `count2`, then processes A and B are not included in the sliced version of the protocol. Since they are the cause of the statespace explosion, the model checker is much faster for the sliced version.

Here is a table containing the number of states generated/time required for different versions of the protocols described above:

	Sliced version	Unsliced version
Simulation example	Terminates	Doesn't terminate
Assertion example	2000000+	305
Never claim example	2000000+	841

It should be noted that slicing as we have described it here does not preserve global properties of protocols. A sliced version of a protocol may no longer contain the components that cause deadlock, for example. However, we believe slicing can be helpful in: understanding protocols when they are in error, reducing

[2] We use the term "step" here to indicate finding immediate dependence predecessors or successors. This is in contrast to a "slice" which transitively follows edges from the immediate predecessors or successors.

simulation time and output, and/or when model-checking a particular protocol is proving to be computationally infeasible. At this point we will describe slicing and, in particular, slicing Promela, in more detail.

Slicing for Concurrent Program Constructs

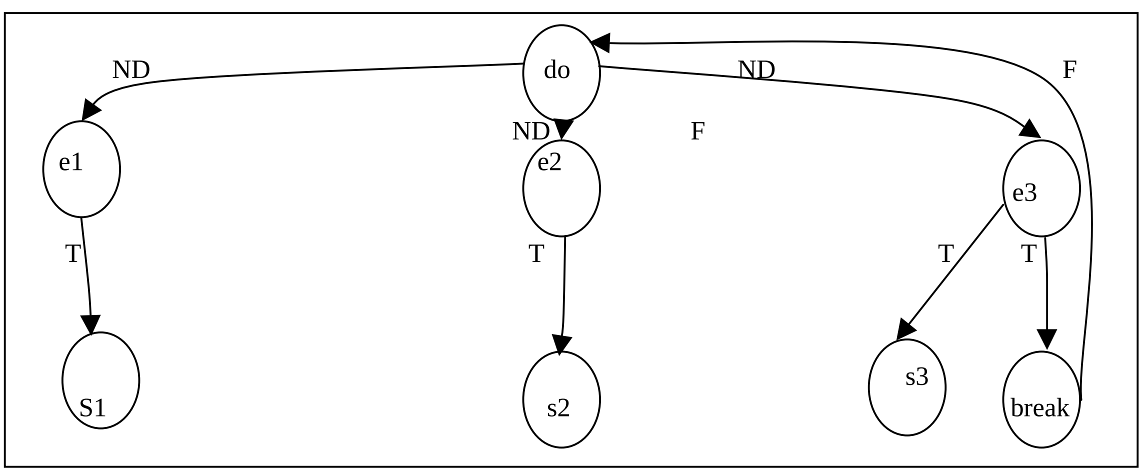
We are extending the program and system dependence graph (PDG/SDG) construction used in the Wisconsin [17] and Grammatech [8] slicers to handle nondeterministic program constructs (`guarded ifs`, `guarded dos`), shared variables (potential interleaving and atomic sequences), and channels as first class objects. The SDG is a program representation that allows the slicing problem to be reduced to a matter of graph reachability. Each node in the SDG represents a statement or condition in the program. The edges signify control and dataflow dependence. Loosely speaking, a vertex is control dependent on a predicate if the predicate controls whether or not the vertex will be executed. A vertex is flow dependent on an assignment if the value assigned can be referenced in the vertex. Slicing with respect to a particular statement involves simply following edges (backward or forward) from the node corresponding to the statement or predicate under consideration. More precise slices can be achieved by employing the notion of context-free language reachability, discussed (with its particular application to slicing) in [16].

We wish to preserve this functionality (that slicing is graph-reachability) while allowing nondeterministic control choices, channel communication edges, and the potential interleaving of statements from concurrently running processes. In this section, we explain how three concurrent Promela constructs are to be dealt with in our extended SDG. We describe how `guarded do` statements will be handled in our revised SDG construction, how shared global variables will be represented in the SDG, and, finally, we discuss some of the issues involved in handling channel communications. We do not go into extensive detail of the required control flow and system dependence graph construction that is required to realize sufficient slicable representations of Promela protocols. The following outlines the sorts of extensions to the SDG framework that are required to handle the concurrent constructs present in Promela.

IF and DO. Promela has both `guarded if` and `guarded do` statements. The `if` in the SDG is represented very similarly to the `do`, so we will discuss only the latter. Consider the following generic guarded DO in Promela:

```
do
      :: e1 -> s1
      :: e2 -> s2
      :: e3 -> s3; break
od
```

The following figure illustrates our planned SDG representation for the above DO statement. Here, edges represent control dependences. Node p is control dependent on node q if the execution of p is contigent on the truth value of predicate q. Note that control *dependence* is not the same notion as control *flow*. We treat the break statement as a pseudo-predicate, as described in [2].



We add a new edge label to the SDG representation: ND, for ‘nondeterministic choice’. (Note that the edge labels T and F represent true and false control dependence respectively.) In a generic slice this new label won’t matter, but edge labels are useful when desiring constrained types of slices, as mentioned previously.

Shared Variables. Promela allows processes to communicate both through channels and through shared variables. Variables referenced in two or more concurrently running processes allow for many data dependence possibilities due to the inevitable different interleavings that may result. As an example, consider the following two process fragments that both reference the global variable `x`:

```
proctype A()                proctype B()
{                           {
        …                           …
        x = y                       q = x
        …                           …
        z = x                       x = r
}                           }
```

A conservative, and therefore potentially imprecise, way to handle this is simply to insert a data dependence edge into the SDG from each definition of a global variable that is not a channel (channels are handled separately, see below) to each use of that variable. (See arrows above.) Thus, in our example, a forward slice with respect to the statement `x = y` will include the statements `z = x` and `q = x`. Moreover, a forward slice with respect to `x = r` will include `z = x` and will also include `q = x`. The latter is due to the fact that we are assuming that process B can run in parallel with itself, hence the assignment to `x` may occur before the use of it when multiple instantiations of process B are running concurrently.

When data and/or communication edges (see below for a discussion of channel communication edges) occur solely due to the assumption that a process may run in parallel with itself, we label those edges accordingly. Ultimately, the user will be able to specify whether to consider those edges when slicing or not. This could be useful when shared-variable interprocess communication between different instantiations of the same process is not of interest.

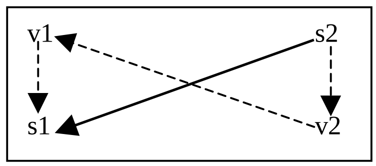
Atomic Statements. Another issue that arises is the use of atomic statement sequences in Promela. Atomic sequences are not interleaved with other code, which eliminates the the need for many data dependence edges that would otherwise exist. For example, in cases like the following (where we assume the atomic sequences are in processes that may run in parallel) the number of data edges can be reduced as illustrated:

```
atomic{         atomic{              atomic{         atomic{
  x = . . .          . . .             x = . . .          . . .
  x = . . .          y = x     →       x = . . .          y = x
  . . .         }                      . . .         }
  x = . . .                            x = . . .
}                                    }
```

In such a case, a reaching-definition analysis can be employed to decrease the number of edges needed in the SDG. Then, only the edge from the last assignment to `x` in the first atomic sequence needs to remain. The advantages to this are twofold: a decrease in the number of edges maintained in the SDG and, hence, increased precision when slicing with respect to statements inside an atomic sequence.

Channels. Slicing a language that has channels as first class objects raises issues not yet addressed explicitly in the slicing literature. Foremost, it is necessary to statically determine, for each channel variable, what other channel variables it may be aliased to at any point in execution of the program. This consideration subsumes several other issues such as global vs. local channel declarations, channels passed as parameters and channels that send other channels as messages. We have developed a theory of channel-possibility sets that will handle these issues [14]. The type of channel analysis that will be performed in a Promela slicing tool is very similar to recent work in pointer analysis. It is an adaptation of the type of pointer analysis discussed in [1].

For the purposes of this paper, we observe that for any given channel variable `c`, a channel analysis as described above describes the other channel variables `c` may send to or receive from. Given this, we can compute something similar to what Cheng calls *communication edges* and add them to the SDG. In Cheng's work, a communication dependence exists between two statements s1 and s2 if there exist two other statements v1 and v2 such that s1 uses a variable defined in v1, s2 defines a variable used by v2 and v2 sends something that v1 receives. Pictorially (dashed arrows represent data dependence and channel sends and receives, the bold arrow is the communication edge):



As an example of where this type of dependence can occur in Promela, consider the following fragments of Promela processes:

```
init
{
      chan x = . . .
      run P1(x)
      run P2(x)
}
```

```
proctype P1(chan c)
{
      p = z
      …
      c!p
      …
}
```

```
proctype P2(chan d)
{
      …
      d?q
      …
}
```

In the above case, our analysis (here, involving straightforward parameter scrutiny) shows that the channel variables `c` and `d` may both represent the same channel constant. Therefore, there is a communication edge from the statement `c!p` to the statement `d?q` signifying that `q` can take on the value of `p`. (This is a slightly different definition from Cheng's notion of communication dependence. In our case, a communication edge from statement s1 to statement s2 represents that information being sent at s1 is (potentially) received at s2. This is analagous to the last condition in the definition of Cheng's communication edges above.) Thus, any backward slice from the statement `d?q` will necessarily include the statement `p = z` (assuming `p` is not redefined subsequently) and, of course, its predecessors in the SDG as well.

Related Work

The slicing literature is extensive. For an introductory survey see Tip [15]. As mentioned in the introduction, Cheng [4, 5] and Krinke [12] have both worked on slicing concurrent programs. Our work is an extension and modification of Cheng's. He devised a theory of program dependence nets which represent more specialized kinds of dependences than we have discussed. Slicing on a program dependence net is, as with PDGs/SDGs, a graph-reachability problem. Cheng's discussion does not take into account dynamic process creation, however.

In order to more precisely handle interleaved code, Krinke moved away from the insistence that slicing be a simple matter of graph-reachability on an appropriate program representation. He devised a notion of threaded program dependence graphs (PDGs) as well as a new notion of dependence termed "interference dependence" in order to more precisely slice concurrent programs. Krinke's algorithm and representation take into account the fact that dependences between statements in parallel processes are not always transitive, and they therefore allow for more precision than both our approach and Cheng's. On the other hand, at present, his algorithm is exponential in the worst case and, as we have shown, even imprecise slices can provide useful information.

We have shown that our techniques may have applications to the model checking community. Slicing can be viewed as a projection, or flattening, of a program. As we have noted previously, however, slicing does not preserve global properties of a protocol. It is very different from, for instance, Holzmann's notion of "protocol generalization" [9] or Shankar and Lam's "protocol projection" [13]. Both of these involve making changes to the protocol before any model checking (or simulation) are attempted.

Generalization is used to simplify protocols before the model checking phase, but unlike slicing, in this process there is an insistence that certain properties be preserved. Holzmann's generalized processes are more general in the way they produce output, so that the behavior of the original protocol is a subset of the behavior of the generalized protocol. This is certainly not the case for a sliced protocol, where a sliced protocol can behave very differently from the complete protocol. Shankar and Lam present a notion of protocol projection that is required to preserve all safety and liveness properties of the original protocol, but is typically easier to analyze. Once again, property preservation is an over-arching concern.

Conclusions & Status

We have demonstrated that, in some cases, slicing can be helpful for protocol simulation, model checking and protocol understanding. Slicing with respect to assertions or never claims that are proving problematic to the model checker will provide insight into what pieces of the protocol are affecting the assertion or never claim under consideration. Slicing can also be useful to decrease the time required for a simulation, by disregarding pieces of the protocol that are not of interest. In general, slicing is useful for debugging, program understanding, maintenance, and testing. Thus, a slicing tool for Promela will be applicable in many ways beyond merely understanding the concurrent constructs, which we have emphasized here.

We have modified the Spin parser to generate a control flow graph in the format required for the Grammatech CodeSurfer software. We have designed the changes that will be needed in the control flow graph file format to account for the concurrent constructs in Promela. It is possible to slice simple Promela protocols that do not use any of the 'new' features we have discussed here. This is all that is needed for CodeSurfer's front end. What is still in progress is the extension to the system dependence graph generation module that will build the concurrent constructs for the SDG correctly from the control flow graph. Once this is finished, it will be possible to slice complete Promela protocols. We then plan to examine how best to generate executable (i.e. simulatable/checkable) Promela slices automatically.

Acknowledgements

Our thanks to Tom Reps and Ed Clarke for many fruitful discussions related to slicing, model checking, simulation, and concurrency. Thanks to David Gries for comments on drafts of this paper and to Paul Anderson for assistance with the Grammatech CodeSurfer software.

References

1. Andersen, L. O. Program Analysis and Specialization for the C Programming Language. Ph.D. Thesis. DIKU, University of Copenhagen, Denmark. May 1994.
2. Ball, T. and Horwitz, S., Slicing Programs with Arbitrary Control Flow. *In Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, (Linkoeping, Sweden, May 1993), *Lecture Notes in Computer Science*, Vol. 749, Springer-Verlag, New York, NY, 1993, pp. 206-222.

3. Binkley, D. Precise Executable Interprocedural Slices. In *ACM Letters on Programming Languages and Systems*, 2(1-4):31-45, December 1993.
4. Cheng, J. Slicing Concurrent Programs – a Graph-Theoretical Approach. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (1993), P. Fritzson, Ed., vol. 749 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 223-240.
5. Cheng, J. Dependence Analysis of Parallel and Distributed Programs and its Applications. In *Proceedings of the 1997 Conference on Advances in Parallel and Distributed Computing*. 1997.
6. Clarke E., Masahiro, F., Reps, T., Shankar S., and Teitelbaum, T. A Program Slicing Tool for Design Automation. Forthcoming.
7. Clarke E. Personal communication related by T. Reps. October 1997.
8. Grammatech. *Software Analysis and Understanding Tools.* http://www.grammatech.com/products/codesurfer/codesurfer.html
9. Holzmann, G. J. Design and Validation of Computer Protocols. (1991), Prentice Hall.
10. Horwitz, S., and Reps, T. The Use of Program Dependence Graphs in Software Engineering. In *Proceedings of the 14th International Conference on Software Engineering* (Melbourne, Australia, 1992), pp. 392-411.
11. Horwitz, S., Reps, T., and Binkley, D., Interprocedural Slicing Using Dependence Graphs. In *ACM Transactions on Programming Languages and Systems* 12, 1 (January 1990), 26-60.
12. Krinke J. Static Slicing of Threaded Programs. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, Montreal, Canada, June 1998, pp. 35-42.
13. Lam, S. and Shankar, A. U. Protocol Verification via Projections. In *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, July 1984.
14. Millett, L. and Teitelbaum T. Channel Analysis and its Relationship to Pointer Analysis. Forthcoming Technical Report.
15. Tip, F. A Survey of Program Slicing Techniques. In *Journal of Programming Languages*, 3(3):121-189, September 1995.
16. Reps, T., Program Analysis Via Graph Reachability. Computer Sciences Department, University of Wisconsin, Madison, WI, June 1998. Submitted for journal publication.
17. Wisconsin Program-Slicing Project. The Wisconsin Program-Slicing Tool, Version 1.0.1. http://www.cs.wisc.edu/wpis/slicing_tool/