Compositional Reachability Analysis Using Process Algebra

Wei Jen Yeh
Michal Young

Software Engineering Research Center
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

State explosion is the primary obstacle to practical application of reachability analysis techniques for concurrent systems. State explosion can be substantially controlled by using process algebra to achieve compositional (divide–and–conquer) analysis. A prototype tool incorporating process algebra is described. The promise and problems of the approach are illustrated by applying the tool to an example that incorporates the alternating bit protocol as a module.

1 Introduction

Among techniques for analyzing the synchronization structure of concurrent systems, enumeration of reachable states in a finite-state model (reachability analysis) is attractive because it is simple and relatively straightforward to automate, and can be used in conjunction with model-checking procedures (e.g., [CES86]) to check for application-specific as well as general properties. Reachability analysis has been used successfully in limited domains like simple communication protocols [Sun81]. Application to real systems has been stymied by combinatorial explosion. This paper describes an approach to controlling the state explosion, and so to making reachability analysis techniques practical for analysis of real concurrent systems.

A *compositional* analysis technique allows one to analyze individual portions of a large system and hierarchically compose partial analysis results. Conventional reachability analysis techniques are not compositional. Process algebras and process calculi [Mil80, Hen88, BHR84, BK84, Hoa85] are compositional by virtue of commutative and associative laws. Algebraic structure can be used to devise a compositional reachability analysis technique. This approach overcomes the state explosion to the extent that one can effectively use a congruence relation [1] to simplify partial products at each stage in the analysis.

This work was supported in part by the industrial sponsors of the Software Engineering Research Center, and by the National Science Foundation under grant 9010135-CCR.

Outline. The paper is structured as follows. Section 2 briefly reviews reachability analysis and then process algebra. Section 3 illustrates the general approach to compositional reachability analysis with a small example. A prototype tool for investigating this approach is described, and in Section 4 it is demonstrated by application to the alternating bit protocol. The alternating bit protocol is then incorporated as a module in the original example to further illustrate the general approach and to demonstrate that state explosion is avoided. Modeling issues and the solutions we have adopted are discussed. Section 5 discusses related work and open problems, and Section 6 concludes.

2 Background

2.1 Reachability analysis

The term "reachability analysis" is used to describe construction of a state-transition model of a system from models of individual processes. The composite state-transition model is often called a "reachability graph." These models highlight synchronization structure and abstract away other details of execution. Reachability analysis has been applied to Petri nets, and CSP-like state machine models, among others [Apt83, Tay83b, Pet81, MR87]. A primary use of reachability analysis is verification of properties of the synchronization structure of software, e.g., freedom from deadlock, freedom from starvation, and freedom from dangerous par-

[1] Strictly speaking, a congruence is not necessary for compositional analysis. It is sufficient to have an equivalence relation such that $p \equiv q$ implies $p \parallel r \equiv q \parallel r$. It need not be the case that $p + r \equiv q + r$, for instance.

allelism. Reachability analysis techniques can also be combined with temporal logic model checking techniques [CES86] to check application-specific properties. With respect to these properties, reachability analysis provides the same level of assurance as formal verification.

The primary obstacle to practical application of reachability analysis for detecting faults in the synchronization structure of concurrent programs is combinatorial growth in the size of the reachability graph. The size of the reachability graph grows as the product of the sizes of individual processes. Moreover, basic complexity results [Lad79, Tay83a, Smo84] imply that there is no universally applicable short-cut. A secondary problem is accuracy, since the details suppressed in building a finite-state model may be essential to the correctness of software. Omitting these details often has the effect of producing spurious error reports, while including more detail exacerbates the state explosion problem.

Compositionality. Conventional reachability analysis techniques are not compositional or incremental. A reachability graph represents a complete, closed system, and is built in a single step. This contributes to the state explosion problem and renders the analysis unusable for large systems. A compositional reachability analysis technique must support a divide–and–conquer strategy wherein reachability graph representations of subsystems can be independently derived and then hierarchically combined to form representations of successively larger parts of a complete system. A compositional approach would also be incremental, since changes to one subsystem would not invalidate the reachability graph representation of another.

The most critical advantage of compositionality is as a lever to overcome the state explosion problem. If the reachability graph representation of a subsystem is transformed into an equivalent but simpler graph, the state explosion problem can be controlled. We must admit that this only partly solves the state explosion problem, since the complexity bounds mentioned above imply that we will not reduce the size significantly in the worst case. Nonetheless we believe that this is can be an effective approach for well-designed systems.

2.2 Process algebra

Process algebra is a widely used framework for describing and reasoning about concurrent systems. A variety of process algebras and calculi have been proposed, among the best known being Milner's CCS [Mil80, Mil89], Hoare's TCSP [BHR84, Hoa85], and Bergstra and Klop's ACP [BK84]. Among the better-known attempts to apply process algebra in practice are the communication protocol specification languages LOTOS [Bri86] and ECCS [CSF89]. Process algebras are growing in popularity due to their relative ease of manipulation and rich abstraction capabilities.

A process algebra consists of a set of action (or event) symbols, a set of operators, and a set of axioms describing properties of the operators. Individual processes can be represented as algebraic expressions involving sequencing and choice among actions, and systems of processes can be described by expressions involving a parallel composition operator. Axioms asserting equivalence among expressions allow one to simplify complex expressions and also to show that an expression describing an implementation corresponds to an expression describing a specification. In particular, it is possible to show that an expression involving parallel composition of processes corresponds to another expression describing non-deterministic sequential behavior.

An expression in process algebra can be modeled as a *process graph*, a rooted directed graph with edges labeled by event names. Parallel composition of process expressions is modeled by a function taking graphs to graphs, and equivalence among process expressions (equality in the algebra) is modeled by an equivalence relation on graphs. Although a reachability analysis system works entirely in the domain of graphs, we can interpret those graphs as models of algebraic expressions provided each graph manipulation models a legal operation on the corresponding algebraic expression. This is what we will mean when we say that a reachability analysis employs algebraic structure.

ACP–η. Bergstra and Klop's Algebra of Communicating Processes (ACP) [BK84] is a simple and general representation for finite-state asynchronous systems with two-party rendezvous. Baeten and Glabeek's ACP–η [BvG87] is a variation on ACP that further simplifies the process composition operation and makes hiding internal details of a subsystem particularly convenient. These considerations led us to choose ACP–η as an appropriate process algebra for investigation of compositional reachability analysis. Space permits only a superficial description of ACP–η here; the interested reader should refer to [BvG87] and [BK84].

In ACP–η, a description of a single process is formed by connecting event symbols together with the sequencing operator ';' and the choice operator '+'. Processes interact in rendezvous fashion by matching events. We adopt the notational convention that an uncomplemented symbol a represents sending a message (or an entry call in Ada), while its complement $-a$ represents receipt of a message (or an Ada *accept*). There is only one choice operator; internal choice is modeled by making a commitment with a silent step η, e.g. η;x + η;y.

In ACP–η the parallel composition operation '$\|$' is defined in terms of two simpler operations, left–merge

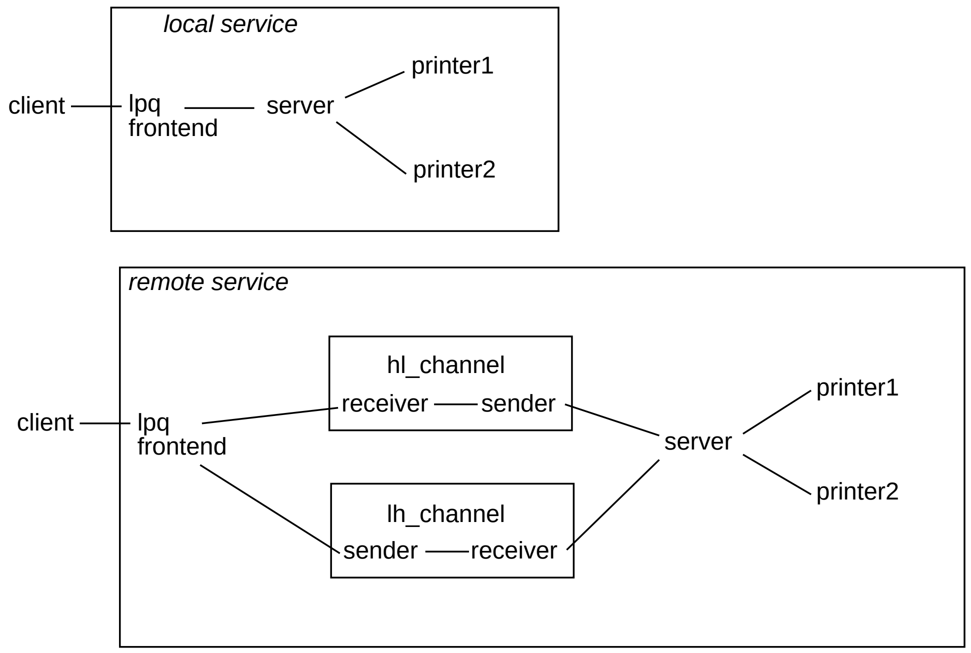

Figure 1: Local and remote versions of a line printer status service

'$\lfloor\!\lfloor$', and communication–merge '$|$':

$$x \parallel y = x \lfloor\!\lfloor\, y + y \lfloor\!\lfloor\, x + x \mid y$$

Informally, this means that when x and y are executed together, either x moves first ($x \lfloor\!\lfloor\, y$) or y moves first ($y \lfloor\!\lfloor\, x$), or they make the first move together ($x \mid y$). Synchronization is by matching an event and its complement, e.g., $a; x \mid -a; y = [a]; (x \parallel y)$. The result of a synchronized move (which we have denoted $[a]$) can be abstracted to the silent action η.

Equivalence among processes can be checked by constructing a rooted η–bisimulation [BvG87] between graph representations of the two processes. Efficient algorithms for this check are known [KS90].

3 Algebraic reachability analysis

3.1 General approach

Reachability analysis based on an algebraic model can use the associative law to permit division of a large system into natural subsystems. It can use restriction and hiding operations in the algebra (analogous to scope rules in a programming language) to isolate internal details of subsystems, and algebraic identities to stem the state explosion.

The general approach is illustrated by the following example. Consider a program that checks and reports the status of either of two printers. This service may be implemented locally, or it may be implemented by communication over unreliable lines to a remote server (Figure 1). We may wish to specify that, from the perspective of a client, behaviors of the remote and local services differ only in speed.

The remote version of the service may be composed of many processes, perhaps more than can be accommodated by a monolithic reachability analysis. Our goal is to suppress details of subsystems when composing. For instance, communication in one direction between local stub and remote server may be implemented by a pair of processes following some well-defined protocol. If we can show that their composite behavior is equivalent to a simpler process (e.g., a bounded buffer), this simpler process can be substituted for the original tasks in the next stage of analysis. Other subsystems would be aggregated and then simplified in a comparable manner. Finally, a representation of the remote service would be produced by composing a few subsystems, and the result would be shown to be equivalent to the local version.

At no point must we contend with the complexity of a complete system, so in principle there is no bound on the size of the system that can be efficiently analyzed. (In practice, we depend on clean modular decomposition and clever specifiers to produce subsystems that can be described by interfaces much simpler than their internal workings.) To achieve this level of divide-and-conquer, we require that our reachability graphs have algebraic structure.

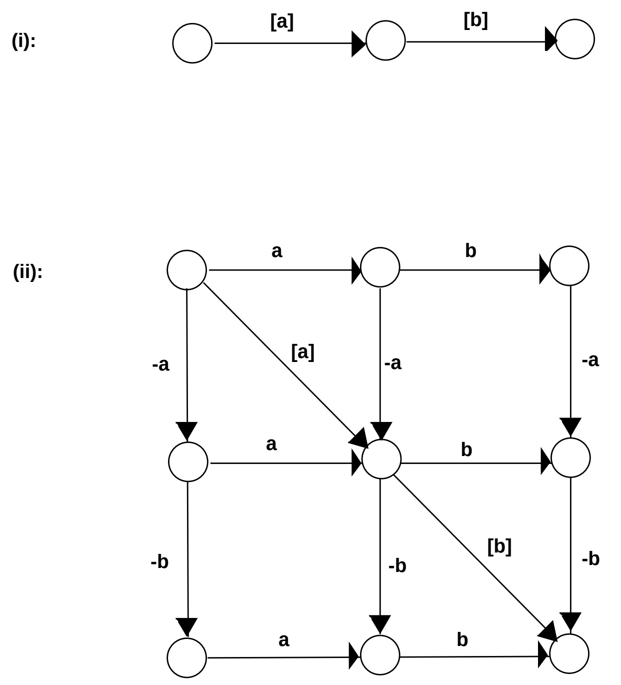

Figure 2: State graph of $a;b \mid\mid -a;-b$ using (i) conventional reachability graph construction and (ii) algebraic product operation

Algebraic structure in reachability analysis. Small modifications to a conventional reachability graph construction suffice to imbue it with algebraic structure. To respect the associative law, it is necessary to consider not only actions taken by the (sub-)system being analyzed, but also the potential for cooperation with external processes. Consider the trivial system $a;b \mid\mid -a;-b$. A conventional reachability analysis will explore only the joint actions $[a]$ and $[b]$. An associative composition operation produces a more complex expression (and graph) capturing the potential for a third process to interact with these. Figure 2 illustrates.

This simple example shows that algebraic structure has a cost; we may generate larger process graphs than would be generated by a straightforward reachability graph construction. While producing larger graphs may seem a step in the wrong direction for overcoming the state explosion problem, it is the price we must pay for compositionality. Moreover, with a bit of extra bookkeeping, restriction (scope wall) and hiding operations can minimize the extra growth. Figure 3 gives the sizes of process graphs encountered during analysis of the line printer status example of Figure 1.

3.2 A prototype analysis tool

We have constructed a prototype tool to evaluate and refine a divide-and-conquer strategy for reachability analysis. This exercise is part of a larger effort to construct a toolkit (the Concurrency Analysis Tool Suite, or CATS) for analyzing concurrent software, and to embed that toolkit in a software development environment [YTFB89]. Initially CATS will be configured for analysis of concurrent Ada programs automatically translated into the task interaction graph model devised by Long and Clarke [LC89]. The prototype described here is designed for rapid evaluation and modification, rather than direct incorporation in CATS. The lessons learned in this effort will be used to redesign parts of CATS to take advantage of the algebraic approach.

Our prototype consists of an Ada-like program design language, PAL, and a processor for that language. PAL includes the main Ada tasking constructs, and in particular includes task entry calls with and without the `select/or delay` construction, and `accept` statements with and without `select.` Guarded accept alternatives and task initiation and termination are notably absent in the current version. No attempt has been made to incorporate Ada features that are not directly involved in tasking. In particular, PAL has no packages or procedures.

The PAL processor consists of three programs. Two of these translate PAL code into a set of process graphs (one graph for each task) along with scope information and analysis directives. The analysis back end builds up a process graph representation of the whole system. The back end simplifies partial products using axioms of ACP-η, and can also test η-bisimilarity of process graphs. The PAL processor performs these tasks automatically or under user control. The current implementation supports all PAL features described in the following sections.

4 Application

Compositional reachability analysis using process algebra is illustrated in this section by application to an example. First an application of the PAL processor to the familiar alternating-bit protocol is described, and then incorporation of that protocol in a larger system. The sizes of process graphs constructed at each step show that (at least for this somewhat contrived example) an exhaustive analysis can be completed without state explosion. The example is described first at a high level, with detailed discussion of modeling issues and their relation to features of PAL held till the final subsection.

4.1 Alternating bit protocol

The main features of the Ada-like PAL language and its processor are illustrated by application to the alternating bit protocol [BSW69]. The example was chosen primarily for its familiarity; the alternating bit protocol is a standard example for protocol specification and verification systems. Among the many treatments of this protocol in the literature, the reader may be interested in comparing [CES86, SMS82, ACW90, LM87].

We write two descriptions of the alternating bit protocol and direct the PAL processor to demonstrate their equivalence. A black box view of external behavior serves as a specification for a white box view of its internal workings. The black box view is equivalent to a single-cell buffer:

```
generic task ABP
   MSG: constant;
   task Client is
      type MSG_Packet is (MSG);
      entry Result (x: MSG_Packet) has no body;
   end Client;
is
   type MSG_Packet is (MSG);
   var m: MSG_Packet;
   entry Input(x: MSG_Packet);
begin
   loop
      accept Input(m);
      Client.Result(m);
   end loop;
end ABP;
```

The meaning of this and the following PAL source code should be clear to readers with some knowledge of Ada, despite the liberties we have taken with syntax (e.g., Ada has no generic tasks). The purposes of the more peculiar features are discussed below in Section 4.3.

In the white box view, we model the alternating bit protocol by a pair of tasks,[2] `Sender` and `Receiver`. It is usual to model the communication medium by a third task, and indeed we could have done so, but in PAL it is more natural and convenient to account for dropped and scrambled messages by non-deterministic choices in the `Sender` and `Receiver` tasks themselves.

```
generic task ABP
   MSG: constant;
   task Client is
      type MSG_Packet is (MSG);
      entry Result (x: MSG_Packet) has no body;
   end Client;
is
   UNKNOWN: constant;
   ACK, NAK: constant;
   ZERO, ONE: constant;
   type Packet is (UNKNOWN, ACK, NAK, MSG);
   type AB is (ZERO, ONE);
   subtype ACK_Packet is Packet range ACK..NAK;
   subtype MSG_Packet is Packet range MSG..MSG;

   entry Input renames Sender.Input;

   task Sender is
      var m: MSG_Packet;
      var a: ACK_Packet;
      var bit: AB;
      entry Input(x: MSG_Packet);
      entry Ack(x: ACK_Packet; bit: AB);
   begin
      loop
         -- Phase ZERO
         accept Input(m);
         loop
            select
               Receiver.InPort(m, ZERO);
               select
                  accept Ack(a, bit);
                  if <<packet not scrambled>> then
                     case a is
                     when ACK =>
                        case bit is
                        when ZERO => exit;
                        when others => null;
                        end case;
                     when others => null;
                     end case;
                  else
                     null;
                  end if;
               or delay 10;
               end select;
            or delay 10;
            end select;
         end loop;
         -- Phase ONE is similar ...
      end loop;
   end Sender;

   task Receiver is
      var m: MSG_Packet;
      var bit: AB;
      entry InPort(x: MSG_Packet; bit: AB);
   begin
      loop
         -- Phase ZERO
         loop
            accept InPort(m, bit);
            if <<packet not scrambled>> then
               case bit is
               when ZERO =>
                  Client.Result(m);
                  select
                     Sender.Ack(ACK, ZERO);
                  or delay 10;
```

[2] The communications protocol literature uses the term "task" in the sense of service specification; we use "task" in the sense of an Ada task, i.e., a process.

```
                    end select;
                    exit;
                 when ONE =>
                    select
                       Sender.Ack(ACK, ONE);
                    or delay 10;
                    end select;
                 end case;
              else
                 case bit is
                 when ZERO =>
                    select
                       Sender.Ack(NAK, ZERO);
                    or delay 10;
                    end select;
                 when ONE =>
                    select
                       Sender.Ack(NAK, ONE);
                    or delay 10;
                    end select;
                 end case;
              end if;
           end loop;
           -- Phase ONE is similar ...
        end loop;
     end Receiver;
begin
   null;
end ABP;
```

The PAL translator represents the `Sender` and `Receiver` tasks by graphs of 14 and 17 nodes, respectively.[3] The composition of these graphs initially produces a new process graph of $17 \times 14 = 238$ nodes.

Scope structure of the PAL code presents the first opportunity for simplification. Entries `InPort` and `Ack` are accessible only within task `ABP`. The PAL translator recognizes this and directs the analysis back end to remove unjoined actions involving these entries (a *restriction* operation in the algebra). The graph is immediately reduced from 238 to 67 nodes, and additional automatic simplifications using identities in the algebra reduce it to 10 nodes.

At this point we instruct the analysis back end to verify the graph by comparing it to the graph representation of the specification (the black box view). The PAL processor demonstrates equivalent synchronization structure by finding a bisimulation between them. (Verification of value transmission is described in Section 4.3.) Having completed analysis of the module, we may substitute the specification graph (2 nodes) for the 10 node process graph.

[3]We will characterize the sizes of process graphs by counting nodes. The number of edges is typically 2–4 times as large.

4.2 Building on: Remote printer status service

Consider again the remote printer status service described in Section 3.1. Having verified that our model of the alternating bit protocol acts as a reliable channel, we can use it as a component in this larger system. In this manner we can build a complete process graph representation for the remote service and demonstrate that it is bisimilar to the local service.

We repeat the composition and simplification steps as for the alternating bit protocol, working bottom-up through a hierarchy of modules. The verification step (bisimulation) is repeated when specifications are available (and often enough to prevent uncontrolled growth of the state space). For the remote printer status example, we provided specifications only for the communication channels and for the system as a whole. We also inserted scope walls at two additional points to aid the PAL processor in reducing process graphs. Figure 3 shows the hierarchy of modules and the number of nodes in each of the process graphs produced (`Scope1` and `Scope2` being the artificially introduced scope walls). The important thing to notice is that the size of the graphs does not always increase as we move up the tree.

4.3 Problems and solutions

Our description of the analysis of the alternating bit protocol glossed over several difficult issues, as well as several features of PAL with which we address those issues. In particular, we must look more closely at how PAL represents data values and data-dependent decisions.

Internal and external choice. Reachability analysis is impractical when data values are modeled in complete detail, so schemes for deriving finite-state models from real programs (e.g., the reduced flow graphs of Taylor [Tay83b] and the task interaction graphs of Long and Clarke [LC89]) generally ignore data values and model data-dependent decisions as internal nondeterminism (arbitrary choice). Translation rules for the core PAL language take the same approach: the `if <<packet not scrambled>>` statement in the example is translated as internal choice and the Ada `select` is translated as external choice.

Value transmission. Even if we limit our analysis to synchronization structure, a model that ignores all data values may be inadequate. The `Sender` and `Receiver` tasks illustrate two situations in which some information about data values must be maintained. We must distinguish between the `ONE` and `ZERO` bits to accurately model coordination of the two tasks, and we must maintain some representation of messages to show that they

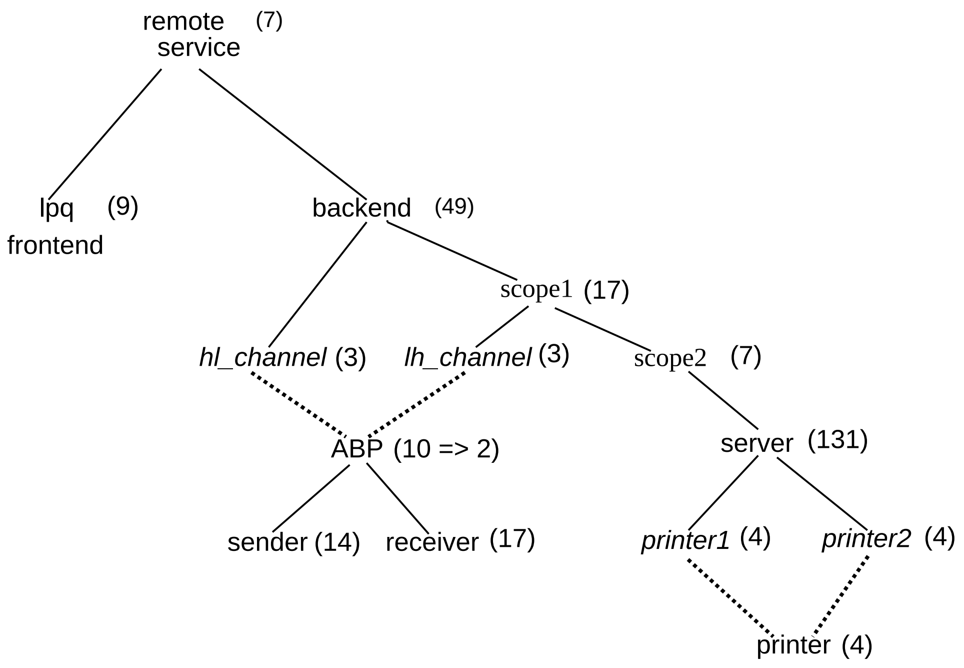

Figure 3: Hierarchy of modules in the remote line printer status example. Numbers indicate the number of nodes in the process graph representation of a module. `Scope1` and `Scope2` are scope walls that correspond to no natural module; they aid the PAL processor in simplifying graphs. Dotted lines indicate multiple instantiations of identical tasks.

are faithfully transmitted.

Value transmission can be modeled by introducing a new event name for each possible value. For instance, if there are two possible values of `m`, then we could replace

```
accept Input(m);
Client.Result(m);
```

by

```
select
    accept Input_m1;
    Client.Result_m1;
or
    accept Input_m2;
    Client.Result_m2;
end select;
```

Of course, a user should not be expected to perform such a transformation at the source level, but this is essentially how the PAL processor forms the process graph representation of a task that transmits values of enumerated types.

It is usually desirable to delay the transformation described above as far as possible. The PAL description of the alternating bit protocol is parameterized by a type `MSG`, which can be replaced by any set of values. Parameterization shields lower level modules from details of their clients and allows them to be independently verified. In our example the protocol is instantiated twice, once for transmission of printer names from client to server, and once for transmission of printer status from server to client. Replacement of `MSG` by its possible values expands `hl_channel` and `lh_channel` only slightly, from the final 2 nodes of `ABP` to 3, but a larger set of values would cause a correspondingly larger expansion factor.

The verification described in Section 4.1 involved demonstrating a bisimulation between the implementation of the alternating bit protocol and its specification before expanding type `MSG` into a set of possible messages. In this form only the synchronization structure is verified. To show that the sequence of messages received is the same as the sequence transmitted, we could instantiate the specification task and each task in the implementation with the actual set of messages before finding a bisimulation.

Better, we can use the property of data independence introduced by Wolper [Wol86] to verify correct data transfer for arbitrary sets of possible messages. We need only verify the protocol with three distinct values to ensure correct data transfer with arbitrary sets of values. This verification was performed separately from the exercise described above; the number of nodes produced was initially 1710, reduced to 197 by restriction, and then to 26 by identities. The bisimulation demonstrated between this graph and a similarly instantiated specification (4 nodes) fully justifies using the initial 2 node representation in the original analysis.

Control variables. Values used for internal control present a slightly different problem. Ignoring them results in an excessively pessimistic analysis. The `case` construct in PAL is used to branch on specific values of an internal control variable. The data independence property does not apply to a variable used in a `case` construct. It is not meaningful to parameterize a task by a variable used in this manner, since its behavior depends on the exact value of the variable. In fact, the `case` construct has no representation in the algebra and is never directly represented in a process graph; the PAL translator unfolds the process graph structure to eliminate it.

Convenience features. The remaining PAL constructs that do not correspond to Ada constructs were added for convenience and efficiency, particularly given the fact that the same language is being used to express specifications and to describe implementations. Parameterization of tasks by other tasks (e.g., the `Client` task parameter to `ABP`) is found in other algebraic systems, and is also roughly similar to generic instantiation in Ada. The `has no body` clause instructs the translator that rendezvous on a particular entry can be represented by a single handshake rather than a begin/end pair. If this clause were not attached to the `Result` entry of the `Client` task parameter to `ABP`, the process graph would have 3 nodes instead of 2.

5 Discussion

5.1 Related work

Reachability analysis techniques of various kinds have been around at least since the 1960's, and attempts to control state explosion are nearly as old. The most common approach in practice is to generate only a sample of the complete state space of a model, which amounts to a kind of testing. Here we mention only a few closely related strands of research regarding process algebra and compositionality.

LOTOS and ECCS. LOTOS [Bri86] and ECCS [CSF89] are extended versions of Milner's CCS [Mil80]. Both were developed for specification and verification of communication protocols, but are similar in broad outline to the PAL system described here. ECCS is similar to LOTOS and is provided with a complete algebraic axiom system. LOTOS is used primarily as a specification and prototyping language, while the verification methodologies associated with ECCS are primarily manual proof and simplification.

LOTOS and ECCS are semantically rich specification languages. PAL is closer to a programming language (Ada), not only in surface syntax but in the way issues like scoping are handled. LOTOS and ECCS provide explicit hiding operations, whereas PAL implicitly hides internal details at scope walls. This is appropriate because the purpose of PAL is to investigate strategies for reachability analysis of real programs.

Concurrency Workbench. The Concurrency Workbench [CPS91, CPS90] supports a variety of manipulations of process graphs, including both bisimulation checking and more satisfactory (but more expensive) preorder checking. Although the Workbench does not directly address issues of modular verification of large systems (for instance, it too requires explicit hiding operations), it does provide a broad and well-integrated set of verification capabilities. One can envision replacing most graph manipulations in the PAL processor by interfaces to the Workbench or a similar analysis engine.

Petri net models. Mandrioli et al. [MZGT85] described a translation from Ada tasking programs to Petri nets as a way of giving a precise semantics to Ada tasking, and Shatz et al. [SMBT90] have used a similar translation to take advantage of existing Petri net analysis tools. The theory and particularly the tools associated with Petri nets are more mature than process algebras, but conventional Petri net reachability analysis is not compositional (construction of the reachability graph of a net is not a commutative and associative operation). Recently first steps have been taken toward providing Petri nets with algebraic structure [MM90, Win87].

SPANNER. SPANNER [ACW90] is a reachability analysis tool for cooperating processes based on *selection/resolution* (S/R) model. In contrast to the two-party rendezvous in PAL, multi-party cooperation in the S/R model makes it relatively easy to instrument a model with "monitor" processes. Monitor processes can carry fairness assumptions and fairness obligations in the form of states that must appear infinitely often in each fair history. While the S/R model has some algebraic structure (composition of processes is modeled as a tensor product), the analysis described in [ACW90] does not exploit it to control state explosion.

PAL uses an algebraic identity of ACP-η to express a simple fairness assumption, but this is less flexible than the approach described in [ACW90]. In principle it should be possible to adapt the methods of SPANNER, but this has not been attempted yet.

Compositional model checking. Clarke, Long, and McMillan have given very general rules for showing that reducing one state transition model to another (as by a bisimulation) preserves properties expressed in a program logic [CLM89]. This framework is particularly attractive if one wishes to specify a system by a combination of process algebra (perhaps disguised as a program, as in PAL) and temporal logic.

Symbolic model checking. In many cases a symbolic representation of a transition relation is far more compact than an explicit enumeration of nodes and edges in a process graph. Burch et al. [BCM$^+$90] have used binary decision diagrams (BDD) to combine symbolic representations of finite-state processes with formulas in the μ-calculus, which in turn can represent formulas of propositional temporal logic. The symbolic form of model checking is in principle subject to the same complexity bounds as conventional reachability analysis, and the practical complexity is very sensitive to the choice of BDD encoding, but symbolic models of digital hardware systems with over 10^{20} states have been constructed and checked. The symbolic approach retards state explosion but does not entirely avoid it; beyond some limit even very clever symbolic encodings will require a divide and conquer approach.

Real-time issues. Our analysis approach ignores real time issues. All delays are modeled using the internal η action, which means rates of progress are treated as being entirely unpredictable. In the PAL construct `or delay` N, the value of N is ignored. Liu and Shyamasundar [LS90] have recently described a reachability analysis technique that does explicitly take time into account. Their model is essentially synchronous; both actual execution and waiting are considered as events that take measurable time. The additional state explosion that would be expected by adding time to the model is controlled somewhat by adopting an unrealistic "maximum parallelism" model, in which each process has its own processor.

5.2 Future work and open problems

A chief limitation of PAL is that it does not support induction over the number of (identical) processes. Bisimulation is too fine an equivalence relation for induction in practical cases such as arrays of tasks or a bounded buffer of arbitrary size. Failure equivalence [BHR84] (or testing equivalence [Hen88]) is better suited to support inductive reasoning, but expensive to test [KS90] and poorly suited for reasoning about fairness. Since there is a natural hierarchy of equivalence relations for process algebras [Bro83], a possible direction for extending PAL is to support multiple equivalences with different granularity.

The PAL processor currently models value transmission by replacing values with unique actions. This is acceptable for small value domains and for verifying modules with the data independence property characteristic of communication protocols, but it is impractical for more general classes of programs. In future we intend to investigate modeling value transmission directly in the underlying algebra.

The PAL processor currently maintains dual descriptions of a program to be analyzed (one serving as specification for the other), but our experience reveals a need for better management of the correspondence between specification hierarchy and implementation hierarchy. We at first naively believed that the two hierarchies could mirror each other, but it is now clear that a more sophisticated correspondence and better support for planning an analysis strategy is required.

We intend to modify a toolkit for analyzing concurrent programs to use the compositional analysis approach described here. This will raise some additional problems, among them maintaining a clear association between the results of analysis and the original source code through several passes of rewriting. Additionally we must give more thought to suitable specification formalisms. Currently our CATS tools [YTFB89] support temporal logic model checking, whereas the algebraic approach suggests using a single formalism for specification, design, and implementation.

6 Conclusion

State explosion is the primary obstacle to practical application of reachability analysis for detecting faults in the synchronization structure of concurrent programs. Conventional reachability analysis techniques are global, but compositionality can be achieved by imposing algebraic structure on the reachability graph construction. The additional cost of representing unjoined actions in process graphs is more than repaid by opportunities for applying reductions.

We have illustrated the compositional, divide-and-conquer approach by application to an example incorporating the alternating bit protocol as a module. Exhaustive analysis (verifying equivalence to a simpler specification) was accomplished without cumulative growth in the state space.

Although work remains to bring reachability analysis techniques to a state of practical utility, the state explosion problem need not be an absolute barrier to exhaustive analysis of realistic software systems.

References

[ACW90] S. Aggarwal, C. Courcoubetis, and P. Wolper. Adding liveness properties to coupled finite-state machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339, April 1990.

[Apt83] Krzysztof R. Apt. A static analysis of CSP programs. In *Proceedings of the Workshop on Program Logic*, Pittsburgh, PA, June 1983.

[BCM$^+$90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[Bri86] Ed Brinksma. A tutorial on LOTOS. In M. Diaz, editor, *Protocol Specification, Testing, and Verification, V*, pages 171–194. Elsevier, 1986.

[Bro83] Stephen D. Brookes. On the relationship of ccs and csp. In *Automata, Languages and Programming (10th Colloquium, Barcelona)*, pages 83–96. Springer-Verlag, 1983. *Lecture Notes in Computer Science* 154.

[BSW69] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex lines. *Communications of the ACM*, 12(5):260–261, May 1969.

[BvG87] J. C. M. Baeten and R. J. van Glabeek. Another look at abstraction in process algebra. In *Proceedings of the 14th InternationalColloqium on Automata, Languages, and Programming (ICALP)*, pages 84–94, Karlsruhe, Germany, July 1987.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CLM89] E. M. Clarke, D. E. Long, and L. McMillan. Compositional model checking. Technical Report CMU-CS-89-145, Carnegie-Mellon University, School of Computer Science, Pittsburgh, PA 15213, April 1989.

[CPS90] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. A semantics-based verification tool for finite-state systems. In *Protocol Specification, Testing, and Verification, IX*, pages 287–302. North-Holland, 1990.

[CPS91] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Machines*, pages 24–37, February 1991. LNCS 407.

[CSF89] Vincenza Carchiolo, Antonella Di Stefano, and Alberto Faro. ECCS and LIPS: Two languages for OSI systems specification and verification. *ACM Transactions on Programming Languages and Systems*, 11(2):284–329, April 1989.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press Series in the Founddations of Computing. The MIT Press, Cambridge, Massachusetts, 1988.

[Hoa85] Charles Anthony Richard Hoare. *Communicating Sequential Processes.* Prentice-Hall, London, 1985.

[KS90] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

[Lad79] Richard E. Ladner. The complexity of problems in systems of communicating sequential processes. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, pages 214–223, Atlanta, Georgia, April 1979.

[LC89] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the Eleventh International Conference on Software Engineering*, Pittsburgh, May 1989.

[LM87] Kim G. Larsen and Robin Milner. Verifying a protocol using relativized bisimulation. In *Automata, Languages and Programming (Proceedings ICALP '87)*, pages 126–135. Springer-Verlag, Karlsruhe, FRG, July 1987. *LNCS 267*.

[LS90] Leo Yuhsiang Liu and R. K. Shyamasundar. Static analysis of real-time distributed systems. *IEEE Transactions on Software Engineering*, 16(4):373–388, April 1990.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.

[MM90] Jose Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.

[MR87] E. Timothy Morgan and Rami R. Razouk. Interactive state-space analysis of concurrent systems. *IEEE Transactions on Software Engineering*, SE–13(10):1080–1091, October 1987.

[MZGT85] D. Mandrioli, R. Zicari, C. Ghezzi, and F. Tisato. Modeling the Ada task system by Petri nets. *Computer Languages*, 10(1):43–61, 1985.

[Pet81] J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[SMBT90] Sol M. Shatz, Khanh Mai, Christopher Black, and Shengru Tu. Design and implementation of a Petri net based toolkit for Ada tasking analysis. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):424–441, October 1990.

[Smo84] Scott A. Smolka. *Analysis of Communicating Finite State Processes.* PhD thesis, Department of Computer Science, Brown University, 1984. Department of Computer Science Technical Report No. CS-84-05.

[SMS82] Richard L. Schwartz and P. Michael Melliar-Smith. From state machines to temporal logic: Specification methods for protocol standards. *IEEE Transactions on Communications*, COM-30(12):2486–2496, December 1982.

[Sun81] Carl A. Sunshine, editor. *Communication Protocol Modeling.* Artech House, Dedham, MA, 1981.

[Tay83a] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[Win87] Glynn Winskel. Petri nets, algebras, morphisms, and compositionality. *Information and Computation*, 72:197–238, 1987.

[Wol86] Pierre Wolper. Specifying interesting properties of programs in propositional temporal logics. In *Proceedings of the ACM Symposium on Principles of Programming Languages (13th)*, pages 184–193, St. Petersburg, Fla., January 1986.

[YTFB89] Michal Young, Richard N. Taylor, Kari Forester, and Debra Brodbeck. Integrated concurrency analysis in a software development environment. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 200–209, Key West, Florida, December 1989. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).