Towards Scalable Compositional Analysis by Refactoring Design Models *

Yung-Pin Cheng
Dept. of Information and Computer Education
National Taiwan Normal Univ.
Taipei, TAIWAN
ypc@ice.ntnu.edu.tw

Michal Young
Dept. of Computer and Information Science
University of Oregon
Oregon, USA
michal@cs.uoregon.edu

Che-Ling Huang
Chia-Yi Pan
Dept. of Information and Computer Education
National Taiwan Normal Univ.
Taipei, TAIWAN
{yklm,pan}@ice.ntnu.edu.tw

ABSTRACT

Automated finite-state verification techniques have matured considerably in the past several years, but state-space explosion remains an obstacle to their use. Theoretical lower bounds on complexity imply that all of the techniques that have been developed to avoid or mitigate state-space explosion depend on models that are "well-formed" in some way, and will usually fail for other models. This further implies that, when analysis is applied to models derived from designs or implementations of actual software systems, a model of the system "as built" is unlikely to be suitable for automated analysis. In particular, compositional, hierarchical analysis (where state-space explosion is avoided by simplifying models of subsystems at several levels of abstraction) depend on the modular structure of the model to be analyzed. We describe how as-built finite-state models can be *refactored* for compositional state-space analysis, applying a series of transformations to produce an equivalent model whose structure exhibits suitable modularity. The process is supported by a parser which can parse a subset of Promela syntax and transform Promela code into refactored state graphs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification - *formal methods, model checking.*

General Terms

Algorithms, Design, Theory, Verification.

*This work is partially supported by National Science Council, TAIWAN under GRANT NO. 90-2213-E-003-009. This work has also been supported by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0034.

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

Keywords

Refactoring, Compositional Analysis, Promela, CCS

1. INTRODUCTION

Although automated finite-state verification techniques and tools have matured considerably in the past several years, they are still fundamentally limited by the well-known state space explosion problem. A variety of techniques have been developed to mitigate space-space explosion. Nonetheless, approaches to increasing the size of system that can be accommodated in a single analysis step must eventually be combined with effective compositional techniques[14, 3, 7] that divide a large system into smaller subsystems, analyze each subsystem, and combine the results of these analyses to verify the full system.

In practice, compositional techniques are inapplicable to many systems (particularly large and complex ones) because their as-built structures may not be suitable for compositional analysis. A structure suitable for compositional analysis must contain loosely coupled components so that every component can be replaced by a simple interface process in the incremental analysis. Moreover, composing the processes and deriving the interface process must be tractable. Otherwise, we need to recursively divide the component into smaller loosely coupled components until every subsystem in the composition hierarchy can be analyzed. However, an ideal structure seldom exists in practice. Designers often structure their systems to meet other requirements with higher priority. It is impractical to ask designers to structure a design in the beginning for the purpose of verifying correctness.

If it is difficult to prove the correctness of a program as originally designed, one may need to prove the correctness of a transformed, equivalent version of the program. This is a notion known as program transformation, which has been widely studied in the area of functional and logic languages. Here, we apply the idea to transform finite-state models to aid automated finite-state verification. In general, the purpose of our transformations is for obtaining, starting from a model P, a semantically equivalent one, which is "more amenable to compositional analysis" than P. It consists in building a sequence of equivalent models, each obtained by the preceding ones by means of the application of a rule. The rules restructure as-built structures which are not suitable for compositional techniques. The goal is to obtain a transformed model whose structure contains loosely coupled components, where processes in each component can be composed without excessive state explosion. We refer to the process as *refactoring.*

The general approach to refactoring and some refactoring transformations were first described in [2] conceptually with an example (without explicit algorithms of transformations). That work describes the application of refactoring to construct network invariants for systems with parameterized behaviors, where those systems are originally inapplicable to inductive verification. However, the transformations described in [2] were derived on an ad hoc basis. They are unlikely to be automated and applicable for general systems. Here we propose a unified approach to accommodate previous ad hoc transformations, extend refactoring to larger class of systems, provide automated tool support, and focus on the major application of refactoring – *compositional analysis*. We report upon a case study involving the Chiron user interface system, comparing analysis performance with results previously reported by Young et al. [16] and Avrunin et al [1].

In past decades, many approaches have been proposed to address the state explosion problem, such as minimizing overall state space, enumerating states implicitly, or abstracting and compacting models. Unlike those approaches which seek improvement in the fundamental techniques, our approach aims for avoiding state explosion at the level of system structure in a compositional fashion.

This paper is organized as follows. In Section 2, we describe the relation between architecture and composition analysis. In Section 3, we give an overview of refactoring. In Section 4, we introduce the refactoring transformations with simple examples. In Section 5, our tool support for compositional techniques is described. In Section 6, we show the results of applying refactoring to two examples. Finally, we end the paper with related work and conclusions.

2. SYSTEM ARCHITECTURE AND COMPOSITIONAL ANALYSIS

When applying compositional techniques to a system, we must divide a system into several subsystems where these subsystems form a hierarchy. Using that hierarchy, we compose processes in a subsystem and replace it by a simpler process which represents the external behaviors of the subsystem (often called an *interface process*[1]). This process works from the bottom of the hierarchy to the top, until whole system is analyzed. Ideally, state explosion can be avoided in this divide and conquer manner but in practice, compositional analysis often yields no savings in analysis effort.

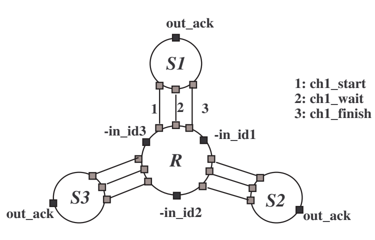
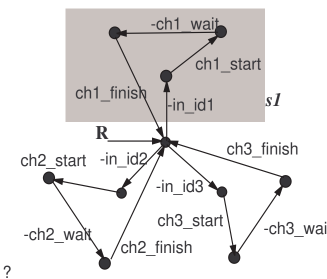


Figure 1: The communication structure of an example system.

Consider an example subsystem in Fig. 1 which consists of processes R, $S1$, $S2$, and $S3$, where $S1$, $S2$, and $S3$ have identical behaviors and R is parameterized by the number of S. In Fig. 2, we show Promela [8] code and state graphs of process R and $S1$. The state graphs are of CCS semantics[12], in which processes communicate by two-way rendezvous. Note that in CCS, paired communications are denoted by a and $\bar{a}$, but we use a and $-a$ instead. In the example, process R iteratively reads an id from channel in (where id is sent by some process which is not in the subsystem) and then uses id to do a sequence of synchronizations with process Si indexed by id. Process Si, after activated by R, tries to send a message ack via a lossy channel out and then return. The ack message is sent to some process which is not in this subsystem. The internal action τ in Si's CCS state graph is to emulate losing message.

```
mtype = { id1, id2, id3,
start, wait, finish }
chan ch1 = [0] of {mtype} ;
chan ch2 = [0] of {mtype} ;
chan ch3 = [0] of {mtype} ;
chan in = [0] of {mtype} ;
chan out = [0] of {mtype} ;

proctype  R () {
mtype id ;
mtype waitmsg ;
do
:: in?id ->
     if
     :: id == id1-> ch1 !start ; ch1?
waitmsg ; ch1 ! finish ;
     :: id == id2-> ch2 !start ; ch2?
waitmsg;  ch2 ! finish ;
     :: id == id3 ->ch3 !start;  ch3?
waitmsg;  ch3 ! finish ;
     fi
od
}
```




```
proctype  S1() {
mtype startmsg, finishmsg;
do
:: ch1?startmsg;
   if
   :: true -> skip ;
   :: true -> out! ack ;
   fi
   ch1! wait;
   ch1? finishmsg ;
od
}
```

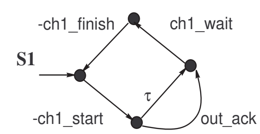


Figure 2: Example process R and S1.

Suppose we want to compose (*R*|*S1*|*S2*|*S3*) in one step. Let *a*={*-in_id1, -in_id2, -in_id3, out_ack*} be the set of ports we must export[2] in the composition. The number of states and transitions generated by parallel composition of (*R*|*S1*|*S2*|*S3*) is 13 states and 18 transitions (see Table 1). After minimized by weak bisimulation, the size becomes 3 states/5 transitions.[3] For larger systems, parallel composition with many processes in one step may suffer state explosion. So, we may try to divide the system and analyze it compositionally. Here, there are few choices to divide the example system. We show three possible subsystems in Table 1, where *b* = {*ch2_start, -ch2_wait, ch2_finish*} and *c* = {*ch3_start,-ch3_wait, ch3_finish*}. Unfortunately, all the subsystems produce state space nearly large as or larger than (*R*|*S1*|*S2*|*S3*). Furthermore, minimizations such as weak bisimulation are much less effective on the state space of these subsystems. Compositional techniques have no merit in this case and sometimes they can even produce worse results [7]. This explains why compositional analysis is thought as a promising approach for combating state explosion but has not yet been widely adopted. In a structure like Fig. 1, no effective subsystems or composition hierarchy can be drawn.

[1]The interface process can be automatically computed from minimizing or abstracting the subsystem state space.

[2]The meaning of exporting ports and restriction operation in CCS are contrary to each other. If a port is exported, it is not restricted in CCS, and vice versa.

[3]The results in Table 1 are computed by Fc2tool[11]
.

Table 1: The state space sizes for different subsystems

	exported ports	*states/ trans*	*minimized*
(R | S1 | S2 | S3)	a	13/18	3/5
(R | S1)	$a \cup b \cup c$	11/14	8/10
(R | S1 | S2)	$a \cup c$	12/16	6/9
(S2| S3)	$b \cup c$	16/40	16/40
(S1|RS1)	d	6/7	3/4

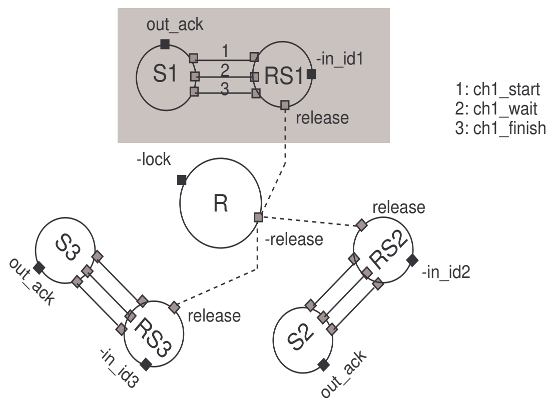


Figure 3: The new structure of refactored example system.

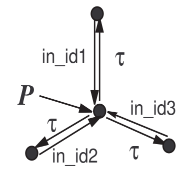


Figure 4: Process P which iteratively invokes *in_idi,i=1 to 3.*

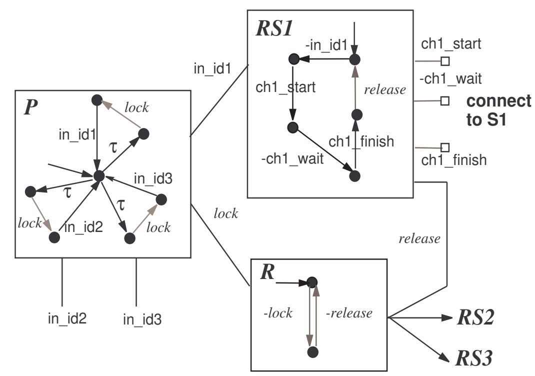


Figure 5: The refactored R,P and the new process RS1.

We say a subsystem is *loosely coupled* to its environment if its interface process contains simple and small state space. So, in a tractable hierarchy, every subsystem must possess such property. Unfortunately, the as-built architectures of many systems do not have this property. In this paper, we propose an approach called *refactoring* to transform a system from an architecture to another with equivalent behavior. For example, refactoring can transform the architecture in Fig. 1 into Fig. 3 by decomposing *R* into *RS1,RS2,* and *RS3*. The behaviors highlighted by shaded region in Fig. 2 are wrapped into a new process *RS1* and the rest is done for *RS2* and *RS3* in similar manner. In the transformation, refactoring creates new synchronizations such as "*-lock*" and "*-release*" to preserve behavioral equivalence and redirects some synchronizations to the new processes. For example, "*-in_id1*" is redirected to *RS1*. In next sections, we will explain the transformations in more detail.

The refactored, new structure of the example system has some good properties which the original structure does not have. For instance, the highlighted region in Fig. 3 becomes tightly coupled inside but loosely coupled outside. The state-space size of (*S1*|*RS1*) is only 6 states/7 transitions (see last row of Table 1, where *d*={-*in_id1, out_ack, release*}).[4] Besides, the behaviors of the subsystem can be minimized more effectively because more rendezvous can be hidden inside the subsystem.

3. AN OVERVIEW OF REFACTORING

To refactor a process, the steps are to decompose its behaviors, make decomposed behaviors into new processes, and redirect communications to the new processes. In the meantime, behavioral equivalence must be preserved.

To explain how refactoring preserves behavioral equivalence, we introduce another process *P* into the example system. *P's* behavior is shown in Fig. 4, which invokes *in_id1, in_id2,* or *in_id3* iteratively and nondeterministically. When *R* is refactored, the shaded region *s1* in Fig. 2 is removed from *R* and wrapped into a new process *RS1*. The refactored behaviors and structure are shown in Fig. 5, where behaviors related to *S2* and *S3* are wrapped into *RS2* and *RS3* in similar manner but are not shown in the figure. After refactoring, *R* becomes a process containing two new synchronizations *-lock* and *-release*. In *P,* the action labeled *in_id1* is now replaced by two actions (*lock.in_id1*) and *in_id1* is redirected to *RS1*. In *RS1,* at the end of *ch1_finish,* a *release* is added to signal *R* the end of an execution cycle in *RS1.*

In principle, the composite behaviors of *P,R,* and *RS1* must precisely simulate the behaviors of *P* and *R*. Let's consider several cases which could happen before refactoring: Suppose *P* invokes *in_id1* and returns. If *P* wants to invoke another *in_idi (i=1 to 3)*, *P* must wait until *R* finishes its sequence of synchronizations with *S1*. So, after refactoring, every *in_idi (i=1 to 3)* in *P* is guarded by a new synchronization *lock.* With *lock,* the new *P* is not able to invoke *in_id1 (i=1 to 3)* continuously. Only after *lock* is granted, new *P* can invoke *in_id1,* which is now redirected to *RS1.* In other words, *R's* new behavior is like a binary semaphore. It makes sure only one *in_idi (i=1 to 3)* in *RSi* is invoked at any time. Thus, the purpose of *release* is obvious. It is used by *RS1* to notify *R* that *RS1* has finished its execution cycle. *R* must be released to allow *P* to invoke another *in_idi (i=1 to 3).*

Mathematically, a behavioral equivalence is needed to justify the transformation. We resort to an equivalence that relates external behaviors of subsystems using weak bisimulation. For instance, we view original (*P*||*R*) as a subsystem because it is changed by the transformation. So, its interfaces are {*chi_start, -chi_wait, chi_finish*}*, i=1 to 3.* Communications like *in_idi, i =1 to 3,* become internal actions of the subsystem and therefore should be restricted. After refactoring, (*P*|*R*) becomes (*P*|*R*|*RS1*|*RS2*|*RS3*). The external interfaces remain the same. The newly added synchro-

[4]The difference of size is not so significant in the example, because it is a very small system. For real applications, the difference can be enormous.

nizations, *lock* and *release,* are internal to the subsystem, therefore, should be restricted. So, in this example, the behavioral equivalence before and after the transformation can be formally expressed as

$$(P|R) \setminus \{in\_idi, i = 1 \; to \; 3\} \approx$$

$$(P|R|RS1|RS2|RS3) \setminus \{in\_idi, i = 1 \; to \; 3, release, lock\},$$

where "\" is the restriction operator and "≈" is the weak bisimulation of CCS. This equivalence can be checked by tools if the correctness of transformations is ever doubted.

As present, we borrow weak bisimulation to justify the correctness of our transformations because it is well-known and supported by several verification tools. Nonetheless, weak bisimulation is not capable of relating two systems for some properties, such as liveness. So, we are working on a new equivalence relation which can precisely relate two systems before and after refactoring. In principle, refactoring does not lose properties like liveness.

Some readers may interest in knowing why we favor CCS over CSP in refactoring. We use the example to explain. It is known that CSP rendezvous is of multi-way rendezvous semantics, which can be formally described as:

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P||Q \xrightarrow{a} P'||Q'}.$$

Suppose there is a process waiting to synchronize with a, it must wait for all other processes which can invoke a. The number of processes participating in the rendezvous may be greater than two.

On the other hand, CCS rendezvous is of two-way rendezvous semantics, which can be formally described as

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}.$$

So, in CCS, if two processes with a want to rendezvous with a communication $\bar{a}$ at the same time, the two processes compete for it. Because of this competition style of rendezvous, *R* can be as simple as that in Fig. 5. That is, the behavior of *R* is independent of other processes. However, in CSP semantics, if we want to have processes compete for some resources, we must introduce more communications to do so. If we adopt CSP semantics, *R's* behavior must be like Fig. 6(A). Its connection structure is shown in Fig. 6(B). The connections between *R* and other processes, unfortunately, grow as the number of *Si* grows. The structure is not as effective for utilizing compositional techniques compared with the one of CCS semantics. In addiction, the structure is inapplicable to inductive verification in [2].

4. REFACTORING TRANSFORMATIONS AND TOOL SUPPORT

To automate refactoring, we adopt Promela as our front-end language and add refactoring commands to its syntax. Promela is a popular design language due to the popularity of SPIN. We select a subset of Promela's syntax (e.g., excluding executable commands like *printf()*) and add some keywords for refactoring. The syntax is called *rc-Promela,* where "*r*" stands for *"refactoring"* and *"c"* stands for *"ccs."* We build a parser in *rc-Promela* syntax to generate CCS state graphs for Promela codes. At present, these CCS state graphs are used as input for Fc2tool[11]. Fc2tool is a toolsuite which can explicitly or implicitly explore state space under CCS semantics. It also provides tools for minimizing and comparing state graphs by weak bisimulation, branch bisimulation, or strong bisimulation.

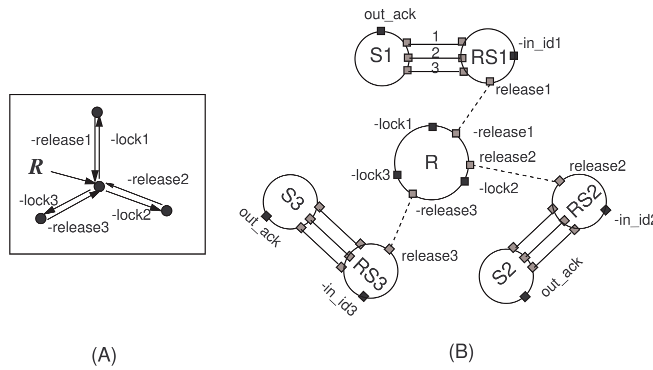


Figure 6: (A) The behavior of R if refactoring adopts CSP semantics. (B) The structure of example system if refactoring adopts CSP semantics.

4.1 rc-Promela and segments

When *rc-Promela* parser is executed, it first creates an abstract syntax tree (AST) for the Promela code. Next, we have an algorithm traverse the AST to generate CCS states and transitions repeatedly starting from an initial state. In the cases without refactoring, when we reach statement "*in?id*," the possible values of variable *id* are "symbolically expanded" to produce three transitions with labels *"in_id1", "in_id2",* and *"in_id3"* and three new states. The new states are put into a queue which saves the unexplored new states. Next, from each new state, one symbolically expanded value of *id* is used to traverse AST. The traversal continues until no more new states are generated and the queue is empty. The CCS state graph in Fig. 2 is so generated from its Promela code in the left.

To activate refactoring, users can use command "**refactorby { }**" to enclose a block of codes. For example, we can refactor *R* by enclosing its Promela codes as follows:

```
proctype R() {
 mtype id ;
 mtype waitmsg ;
 refactorby id {
  do
  :: in?id ->
     if
     ::id ==id1->ch1!start;
       ch1?waitmsg;ch1!finish ;
     ::id ==id2->ch2!start;
       ch2?waitmsg;ch2!finish ;
     ::id ==id3->ch3!start;
       ch3?waitmsg;ch3!finish ;
     fi
  od
 }
}
```

We call the enclosed block as *r-block.* The codes inside an r-block often begin with a statement which creates branches of control flow, such as *do* block followed by a "*in?id*" in this example or *if* statement. Decomposing behaviors at these locations often creates useful segments (defined later) for compositional analysis.

When AST traversal algorithm enters an r-block, the way of generating CCS state graphs is changed. It begins to generate CCS state graphs in *segments.* A *segment* is defined as the states and transitions generated by one pass of an r-block via the AST traversal algorithm. For the example above, starting from the first statement of r-block, which is "*in?id,*" we use *id1,* one of the symbolically expanded values of *id,* to traverse the r-block in one pass and we obtain a segment in Fig. 7.

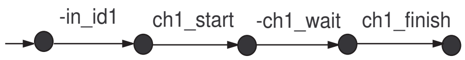


Figure 7: Segment seg_id1.

We call this segment *Seg_id1.* If we use other symbolically expanded values, *id2* and *id3* to traverse the r-block in one pass*,* two similar segments *Seg_id2* and *Seg_id3* are generated for this r-block. We call the first transition of a segment as *guard.*

4.2 Grouping segments

Once segments are generated, the next step is to divide the segments into groups and wrap each group in a new process. For process *R*, we already know *Seg_id1, Seg_id2,* and *Seg_id3* are divided into 3 groups and each is made into a new process (see *RS1* in Fig. 5). The grouping options are specified by the list behind the keyword **refactorby**. In this example, command **refactorby** is followed by a variable name *id,* which instructs the refactoring algorithm to group segments by every possible values of variable *id* (or conversely, divide the segments into groups by the values of *id*). For example, segments whose transition labels contain *id1* (which is symbolically expanded from *id*) are grouped together. Process *R* shows the simplest case of refactoring – one segment in one group. Section 3 already shows how it is transformed. In practice, process's behaviors can be more complicated. After grouping, one group may contain more than one segment. A unified transformation (see section 4.3) is derived to deal with such general cases.

We call the list behind keyword **refactorby** as *grouping options*. Since the segments in a group will be wrapped in a new process, the options decide the number of new processes to be created by refactoring. One mostly used grouping option is variable names like "**refactorby** *var1, var2.*" This option first divides the segments into groups using all possible values of *var1* and next divides these groups again using all possible values of *var2.* Sometimes, for systems such as those in section 6 we need to use channel name and a variable name to divide the segments. The refactoring command is like "**refactorby** *ch, var,"* where *ch* is a channel name and *var* is a variable name. This option first divides the segments into two groups, one containing segments with *ch* in their edge label and none in the other group. Next, refactoring uses all possible values of *var* to divide the two groups into smaller groups.

To allow flexible refactoring decisions to be made, the grouping options can be specified as expression, such as "**refactorby** *ch1 and id == id1, waitmsg*." However, in practical applications we have encountered, most behavior patterns of processes are regular or patterned. So far, complicated options like that have never been used practically.

4.3 The unified decomposition transformation

In practice, behavior patterns to be refactored are often complicated by parameterization and the presence of data values. A variable with a finite range is typically "unrolled" in finite state verification, i.e., each state s in a process may be replaced by s_1, s_2, ... s_n for the n possible values of the variable (or the cross product of multiple variable values). On these states, same request may be responded differently. In Fig. 8, we introduce another process *RR*. *RR* receives an index from channel *in* and uses the index to address an element in array *cv*. Next, *RR* output the value of the element and flip the element's value.

```
mtype = { zero, one }
proctype RR() {
  bit id ;
  bit cv[2] ;
  refactorby id {
  do
  :: in?id ->
     if
     :: (cv[id] == 0) ->
        out ! zero ;
        cv[id] = 1 ;
     :: (cv[id] == 1) ->
        out ! one ;
        cv[id] = 0 ;
     fi
  od
  }
}
```

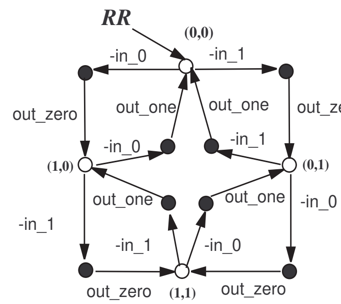


Figure 8: The Promela code and state graph of process *RR*.

When we begin traversing the Promela code to generate state graph, the initial state is set as a product of $(cv[0], cv[1])$, which is initialized as (0,0). The easiest way to represent a state is using the product of all variables plus an address of current statement. There are three variables $cv[0], cv[1]$, and id in this example but id can be excluded from the product because including it in the product is irrelevant for producing state graph. Whether a variable is relevant for producing state graph are checked statically by data flow analysis, but here we ignore the implementation details.[5] Also, in the figure, the address information in the product is omitted.

The state graph in Fig. 8 shows that when id=0 is received from channel *in* at first time, *RR* outputs "zero" and enters a new state (1,0). Next time, when id=0 is received, *RR* outputs "one" and returns to (0,0). So, when you send *RR* a zero, the outputs may vary depending on the state of $cv[0]$; that is, *RR* is a stateful process.

Suppose refactoring is activated. Using *id=0*, we begin the traversal of r-block. At the end of r-block, we produces a segment *Seg0* in Fig. 9. Let the traversal continue from a new state (1,0). We return to the beginning of r-block and use *id=0* again to traverse the r-block. Another segment *Seg1* in Fig. 10 is produced. Suppose the grouping option is "**refactorby** *id*." The two segments belong to the same group.

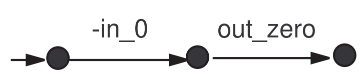


Figure 9: Segment $Seg0$.

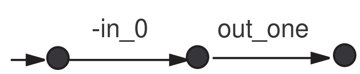


Figure 10: Segment $Seg1$.

[5] The data flow analysis is also used to decide which variable should be represented by a value process, which will be described later.

Consider wrapping *Seg0* and *Seg1* into a new process, say *RRS0,* and redirecting action *in_0* to it. One problem arises – should we redirect to *Seg0* or *Seg1*? We know it is determined by $cv[0]$. To assist *RRS0* making the choice, we introduce a new process called *Value Process* (VP). We use $VP(v)$ to denote the value process of a variable v. In Fig. 11 we show the value process of the variable $cv[0]$.

Constructing a $VP(v)$ for a variable v is straightforward. If there are n possible values of variable v, create n states to represent each value. At each state, add a transition which returns to itself labeled "*-v==#*," where # is the value of the state. Between states, if v can change its value from i to j, add a transition labeled "*-v:=j*" between state i and state j. Next, search in segments the places where v is updated and insert an edge labeled "*v:=#*," where # is the new value *v* is changed to. As of this example, we append "$cv[0] := 1$" to segment *Seg0* and "$cv[0] := 0$" to segment *Seg1*.

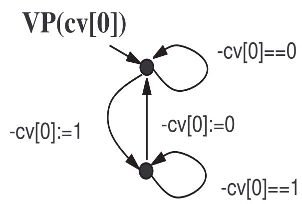


Figure 11: The value process of $cv[0]$

Once $VP(cv[0])$ is constructed, we place *Seg0* and *Seg1* together to create a new process *RRS0* (see Fig. 12). In the beginning, *RRS0* must be enabled by a new synchronization *"-startRRS0.*" This synchronization is to prevent *RRS0* from rendezvousing with $VP(cv[0])$ privately. Next, either "$cv[0] == 0$" or "$cv[0] == 1$" is enabled by $VP(cv[0])$ to activate the correct segment. At the end of segment, $release$ is used to release *RR*. Note that inside the caller of (*in_0*), every (*in_0*) is now replaced by (*lock*•*startRRS0*•*in_0*).

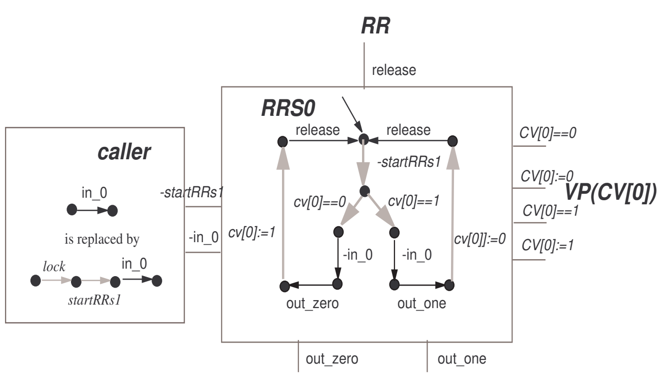


Figure 12: The state graph and interface of $RRS1$**.**

In Algorithm 1, we list the algorithm of this unified transformation. The algorithm has other variants to deal with different kinds of guard in segments, such as τ or else. These variants are not listed in this paper.

Without loss of generality, we assume there is only one state variable v which has n possible values. So, there are n self-loop transitions labeled "*-v==i*" in $VP(v)$. The algorithm assumes there are n segment to be wrapped into a new process. Let the segments be collected in a set α and each segment α_i is activated by transition "*v==i*." Let a be the guards of segments in α. The algorithm also assume transitions labeled "*v:=i"* have been inserted properly. The algorithm is quite straightforward. Its complexity is $O(N)$, where N is the number of segments.

Algorithm 1 The unified decomposition transformation

UnifiedDecomposition(α,a)
begin
  Construct $VP(v)$ for segment in α.
  Create an empty state graph T with an initial state s_0.
  Add transition $\langle s_0, -startT, s_1\rangle$ to T
  for each segment α_i in α do {
    copy α_i to T.
    add transition $\langle s_1, v == i,\ t_i\rangle$ to T, where t_i is the initial state of segment α_i.
    for every exited state s_e of α_i do
      add transition $\langle s_e, release, s_0\rangle$ to T.
  }
  update other processes whose edges labeled "a"
  into $lock.startT.a$.
end.

4.4 Simplifying state graphs

Although the processes in Fig. 12 look more complicated than the original, most synchronizations are internal between $RRS0$ and $VP(cv[0])$. This seemingly complicated synchronizations can be easily conquered by grouping them into subsystems. Fortunately, for many cases, we can transform them into a more compact form. For example, it is not difficult to determine that segments $Seg0$ and $Seg1$ are activated one after another regularly in a loop. Using this observation, we can delete $VP(cv[0])$ and reduce *RRS0* into *RRS0'* of Fig. 13.

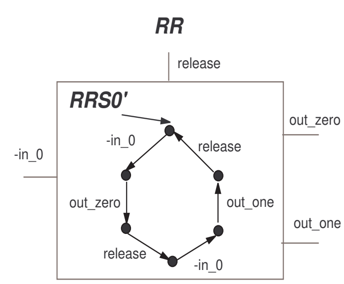


Figure 13: The simplified $RRS0$

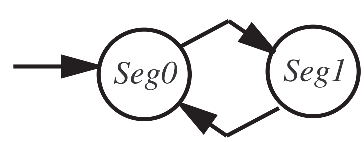


Figure 14: The directed graph of Seg0 and Seg1.

To simplify a new process in this way we need to know whether segments activate other segments in a regular and predictable way. The algorithm is listed in Algorithm 2. The algorithm attempts to construct a directed graph representing the activation relations among segments. Let each node represent a segment. If only a directed edge is established from a segment a to a segment b, it means segment b is activated for next time after segment a is executed. If there are more than one outgoing edges from a to other segments, it means one of those segments can be activated for next time. So, the directed graph of *Seg0* and *Seg1* can be constructed as Fig. 14. From *Seg0* to *Seg1* there is *exactly one* directed edge and vice versa. It means that *Seg0* and *Seg1* activate each other in a deterministic way and looped like the directed graph in Fig. 14. That is, they are eligible for simplification.

Let α be the set of segment from which the new process is constructed and $|\alpha| = n$. Let each segment α_i is activated by the transition labeled "*v==i*." Again, without loss of generality, we assume there is only one control variable v which have n possible values and each value i can activate segment α_i. Let $Pre(s)$ be the set of edges that start from some states and end at s. Let $Src(e)$ be a function which returns the source state of an edge e. Let $Exited(S)$ be the set of exited states of segment S.

Given a segment α_i, we use data flow analysis (procedure *ComputeGotoSet*) to compute the set of segments that could possibly be activated by α_i at every exited states. If a segment α_i can activate a segment α_j, we add a directed edge between node α_i and α_j. Initially, a segment activates itself. So, if v is not updated in the segment, there is at least one outgoing edge back to itself. Once the directed graph is constructed, we check if every node has exactly one outgoing edge. If not, the segments in α are not suitable for the simplification. If yes, we follow the directed graph G to connect the segments into a new process. The algorithm has a low-order polynomial complexity. For the procedure ComputeGotoSet, provided the number of edges into each state is bounded by a constant, the worst-case complexity is $O(S^2)$, where S is the number of states in a segment. So, the worst-case complexity of Algorithm 2 is $O(NS^2)$, where N is the number of segments.

Note that Algorithm 2 is always used to check segments first. If they are eligible for simplification, follow its directed graph to connect segments. If not, Algorithm 1 is used to wrap them into a new process.

Algorithm 2 The simplification transformation

```
Simplification(α)
begin
  Initialize G as an empty directed graph.
  Create a new node t_i for segment α_i in G.
  // construct directed graph G from segments
  For each segment α_i in α do
    goto ←ComputeGotoSet(α_i);
    for each integer k in goto do
      add a directed edge t_i → t_k in G
   if (|goto| == 1) then mark node t_i
  end for ;

  // check the directed graph to see if it is
  // eligible for reduction
  if (all the nodes in G are marked) then
    // the case is eligible for reduction
    Follow G to connect the segments
    into a new process.
  else
    return "not eligible for reduction";
  end if ;
end.

procedure ComputeGotoSet(α_i)
begin
  Let s_0 be the initial state of α_i;
  goto(s_0) = {i}; // initially, a segment activates
            // itself for next time.
  Set out(e) = { } for all the edges of α_i.
  Repeat
   for each state s in α_i do
     for each edge e ∈ Pre(s) do
      if (label(e)=="v:=i") then out(e) = {i};
      else out(e)=goto(Src(e));
     end for ;
     goto(s) := ∪_{e∈Pre(s)} out(e);
   end for ;
  Until goto(s) has no change for all s;
  return ∪_{s∈Exited(α_i)} goto(s);
end.
```

5. TOOL SUPPORT FOR COMPOSITIONAL ANALYSIS

To facilitate compositional analysis, we build a set of tools on top of Fc2tool [11] to automate hierarchical composition. Although Fc2tool provides some support for compositional analysis, it is too tedious and difficult to use directly. For example, to create a hierarchy in Fc2tools, users must create, label, and connect every port by hand, which is error-prone and time consuming.

To use our tools to compose a system hierarchically, a user only needs to put the state-graph files (in a format for Fc2tools) in a directory and provide a hierarchy file like the following:

```
T1 := P R
T2 := S1 S2 T1
@ T2 is the whole system
```

Our tools will compute the necessary information automatically. In the example, suppose there are four state-graph files, *P,R,S1,* and *S2*. When P and R is composed into $T1$, our tools examine the directory and know its environment is constituted by $S1$ and $S2$. Correct label restriction (or exportation) for $T1$ is computed automatically. Unless specified, weak bisimulation is the default method for minimizing the state space of subsystem. In next section, all the experiments are done in this environment with 128M of memory under Linux platform.

6. EXAMPLES

In this section we demonstrate the power of our approach by two examples. We choose the examples by two reasons. First, both examples have been used to gauge the scalability of verification tools. Second, when their system sizes increased, their as-built architectures make compositional techniques futile.

6.1 The elevator system

The model of elevator system is extracted from the elevator system of Richardson et al. [13]. Its implementation uses array of Ada tasks and is designed to be extended to arbitrary number of elevators. If the number of elevators is n, there are $n+3$ tasks, including n $elev\_sim\_task[i]$ which emulate the moving elevators for lifting customers, one *controller* task to command elevators to serve hall calls or car calls, one command dispatcher task (*elevator*), and one task (*driver*) to emulate customer pushing the hall call or car call button. In [5], Corbett analyzed (by global analysis) the system with maximum to 4 elevators. However, due to the difference in analysis tools used, memory capacity, and platform, we can only analyze up to 3 elevators in our environment.

We use elevator index and channel name as grouping options to refactor *controller* and *elevator*. In the model, they both are not stateful processes, so, no VP is created. The model is refactored in a way similar to the steps of refactoring the example in section

3. The number of new tasks created by refactoring plus the original task is listed in Table 2.

Fc2tools can enumerate reachable states explicitly and report the number of explored states and transitions. In a composition hierarchy, among all the subsystems analyzed, we pick the one which consumes most memory as the memory requirement to accomplish the compositional analysis. We compare three methods in Fig. 15. They are global analysis, compositional analysis without refactoring, and compositional analysis with refactoring. The hierarchy to compose the system without refactoring is (..((*controller*|*elevator)*| *elevator_sim_task1)*|..*elevator_sim_taskn)*| *driver).* This hierarchy is carefully chosen by experience and trial-and-error so that state explosion is at least not worse than global analysis.

In the experiment, both global and compositional analysis without refactoring exhaust memory rapidly. On the other hand, the refactored model shows mildly linear growth of memory usage and the ability to analyze hundreds of elevators. The hierarchy for composing the refactored elevator system is to compose a base system with one elevator first and then gradually add other elevators. Weak bisimulation and context constraints [3] are used to reduce subsystems in the hierarchy. One of the reasons that it can be analyzed to hundreds of elevators is that its refactored structure is "near to" a structure that is suitable for inductive verification (see [2]). We only check for deadlocks in this experiment. Because we adopt weak bisimulation, we currently limit our approach to safety properties, where a safety property can be translated into a deadlock detection problem [4].

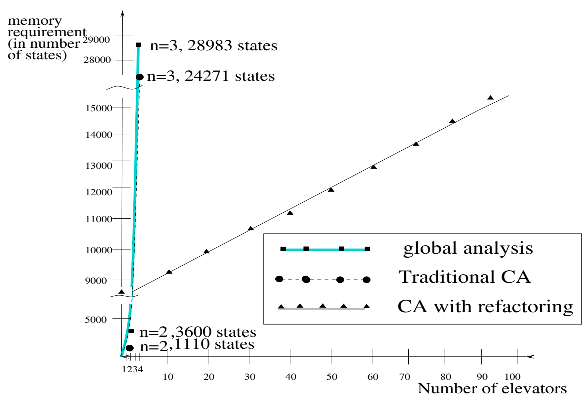


Figure 15: The states generated for elevator system by (1) global analysis (2) compositional analysis (3) compositional analysis with refactored structure.

6.2 Chiron user interface system

Chiron user interface system [10] is a moderate-size concurrent Ada program. It was built to address concerns of cost, maintainability, and sensitivity to changes in the development and maintenance of user interfaces for large applications. Chiron's design philosophy is to separate application code from user interface code. So, there are user interface agents called *artists* attached to selected data abstract types (ADT) belonging to the applications. At runtime, each artist can register *events* of interests to *dispatcher.* Whenever there is an operation call on the ADT, the dispatcher intercepts the call and notifies each of the artists associated with that ADT with the event.

Chiron has been analyzed by Young et al. [16] and Avrunin et al. [1], both with 2 artists analyzed. Its Ada code and Promela code can be accessed via *http://laser.cs.umass.edu/verification-examples*. In [1], different analysis tools (INCA,SPIN and FLAVERS) are stressed by increasing the event number of Chiron model. In that study, they decompose the dispatcher task into a subsystem with a separate task that maintains the array of each event, together with a single interface task that receives the requests for registration, unregistration, and the notification of events and passes them to the appropriate task for a particular event. Consequently, INCA shows better performance than SPIN and FLAVERS. The decomposition resembles our refactoring. However, it is done by rewriting design codes which requires human expertise and is difficult to automate and guarantee behavioral equivalence.

To demonstrate the power of refactoring, our tool is stressed by increasing the number of artists. A 2-artist Chiron consists of 6 tasks. So, for an n-artist Chiron, there are $n + 4$ tasks.

When we began refactoring *dispatcher* task, we begin to realize why the number of artists has been limited to two in the past. For each event, the dispatcher maintains an array for bookkeeping the registered artists. The array is implemented as a queue. For example, suppose there are 3 artists, a_1, a_2 and a_3, registering an event e consecutively. Let the event array be $e[\,] = (a_1, a_2, a_3)$. If a_2 unregisters the event e, the content of $e[\,]$ becomes $(a_1, a_3, \_)$; that is, the artists behind a_2 are shifted left by one element. Assume there are n artists, all possible combinations of the array are $1 + \sum_{i=1}^{n} \binom{n}{i} i!$. If there are m event array, the combinations are $(1 + \sum_{i=1}^{n} \binom{n}{i} i!)^m$. So, task dispatcher grows at a prodigious rate as the number of artists increased. On the other hand, for a 2-artist dispatcher, it contains only 5 combinations. When event number is m, the size of 2-artist dispatcher is proportional to 5^m and the number of tasks remains unchanged.

In our first attempt, the unified transformation constructs a VP for each array element and wraps segments into new processes for different artists. The grouping options are channel name and artist id. Unfortunately, the refactored structure has little hope to scale well compositionally because VPs not only communicate with segments, but also with other VPs. This happens when an artist is unregistered. It starts a cascading changes of VPs because of shifting elements in the event array.

After a second look at the code, we discover that the event array, though implemented like a queue, is actually used only for keeping track of which artists are registered. The unregistration need not obey the FIFO rule. We are not sure why the event array is so implemented. Actually, a bit array of size n is adequate for the bookkeeping. For example, an event array $e[\,] = (1, 0, 1)$ means artists a_1 and a_3 have registered. If an artist wants to unregister the event, dispatcher simply sets its bit to zero. By replacing the queue with this bit array, the size of dispatcher becomes proportional to 2^n. Although 2^n is still a formidable growth rate, refactoring can create VPs and new tasks which are loosely coupled to its environment. We analyze the refactored Chiron with bit array compositionally. It can be analyzed up to 14 artists as in Fig. 16.

Table 2: The summary of refactored elevator model.

task name	no. of states	the number of subtasks after refacting
controller	$21n$	$2n + 1$
elevator_sim_task[i]	8	no change
elevator	$7n$	$n + 1$
driver	$8n$	$n + 1$

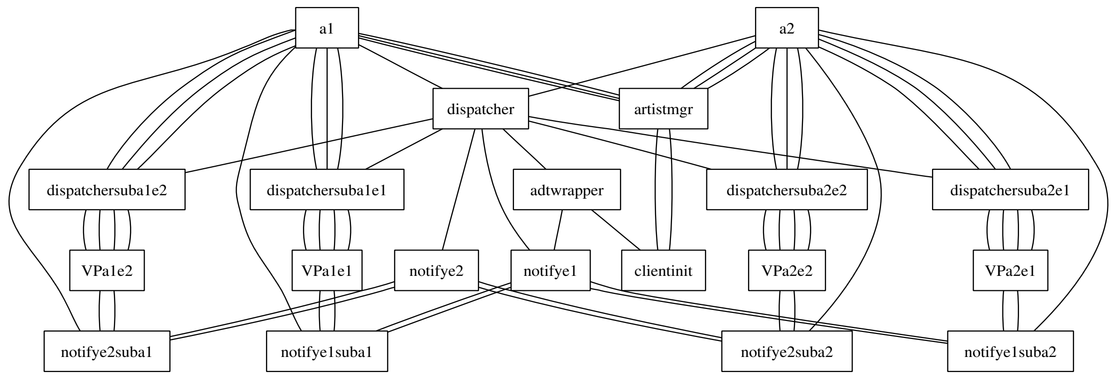


Figure 17: The refactored structure of 2-artist Chiron system.

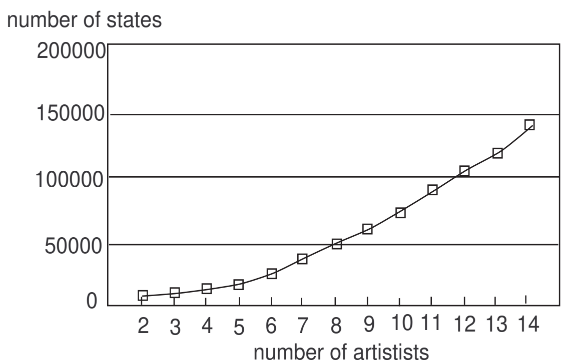


Figure 16: The states generated from refactored Chiron (bit array version).

The refactored structure of a two-artist Chiron is shown in Fig. 17. The dispatcher task is in the middle. The new tasks generated from decomposing dispatcher is named "*dispatchersubaXeY*," where X is the index of artist and Y is the index of event. The structure may look complicated in the beginning. Searching a tractable composition hierarchy seems difficult. However, heuristics exist for searching the hierarchy.[6] We can search in the components of first artist first. Like composing the elevator example, we start by composing all the components of artist *a1*. Let A = (notifye1| notifye1suba1), B = (notifye2 | notifye2suba1), C = (dispatchersuba1e1|VPa1e1), D = (dispatchersuba1e2|VPa1e2) , E = (adtwrapper| clientinit | artistmgr). The composition hierarchy we use is ((((C | a1 | dispatcher) | D) | A| B) | E). After the base system is completed, we begin composing components of artist *a2* into the base system.

Composing the base system always generates maximum states and transitions, because there are transitions and states must be exported for rendezvousing with other artists later. The number of these exportations is strictly decreased when more artists are composed. In the experiment of 15 artists, the weak bisimulation of Fc2tool consumes too much time and disk space while composing the base system.

Although we consider the result a significant improvement, the extent of improvement is not as good as elevator system. The cause is that Chiron contains a FOR loop that makes rendezvous with all artists sequentially. This small part of behaviors can not be refactored perfectly and made into a semaphore independent of other processes. As a result, more exportations are inevitable. Since FOR loop like that can be common in practice, research to address the problem continues. We have found that if the order of making rendezvous in a FOR loop is irrelevant to property of interest, we may refactor the behavior into a better structure for compositional analysis. However, more works remain to be done for that, so we exclude the results from this paper.

[6]Finding an optimal solution is NP-hard, but in practice an acceptable solution can often be found with modest effort.

7. RELATED WORK

Researches addressing systems that are not suitable for compositional, incremental analysis can be found in [15, 6]. Corbett and Avrunin [6] observe that in the analysis of large and complex programs, a module I may not be further decomposed into many loosely coupled units and the composition of its processes may yield intractable results. They assume the module interfacing with its environment is a specification process S. The analysis of the module becomes a problem of proving the trace equivalence between S and I using integer programming techniques. Their approach, however, is limited to a restricted class of systems, requiring processes to be deterministic.

Yeh and Young [15] considered the problem of "design for analysis" with a goal of compositional analysis using process algebra. In that work, it was proposed that analyzability should be an important factor in structuring the design of actual implementations. Be that as it may, one is often in the position of trying to apply analysis post hoc to systems that are not structured as we would like. Even if one has the luxury of structuring a design with analysis in mind, there may be good reasons (performance, physical distribution, reuse of existing artifacts) for the "as built" system to differ from the structure needed for verification, and a way of reconciling the verified and as-built structures will still be needed.

8. DISCUSSIONS AND CONCLUSIONS

In [9], Holzmann has argued that blindly derived design models are unlikely to be amenable to analysis. That argument applies to other verification tools and may apply to refactoring on occasion as well. The array usage as a queue in Chiron is an example. That particular data structure may prevent refactoring from constructing suitable structures for compositional analysis. Fortunately, behaviors of most systems are regular or patterned. The growing popularity in design patterns shall reduce the occurrence of those problematic behaviors. On the other hand, we will continue exploring new transformations for those problematic behaviors and data structures.

Automating the heuristic search of a tractable hierarchy is another problem worth pursuing. Furhter automation is important for producing refactoring tools that can be used routinely by software developers.

In summary, the refactoring transformations described here permit decomposing processes and recombining them in a structure that is more amenable to compositional analysis using process algebra, allowing larger versions of a model to be verified. We have described some important refactoring transformations and a refactoring tool that automates restructuring models in a subset of Promela.

9. REFERENCES

[1] G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Pasareanu, and S. F. Siegel. Comparing finite-state verification techniques for concurrent software. Technical Report UM-CS-1999-069, Dept. of CS, University of Massachusetts, November 1999. (in preparation).

[2] Y. Cheng. Refactoring design models for inductive verification. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA2002)*, pages 164–168, Rome, Italy, July 2002.

[3] S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, October 1996.

[4] S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8:49–78, January 1999.

[5] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 2(3):161–180, March 1996.

[6] J. C. Corbett and G. S. Avrunin. Towards scalable compositional analysis. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering.*, pages 53–61, New Orleans, Louisiana, USA, December 1994.

[7] S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Proceedings of the 2nd International Conference of Computer-Aided Verification*, pages 186–204, 1990.

[8] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, NJ 07632, 1991.

[9] G. J. Holzmann. Designing executable abstractions. In *Proceedings of the second workshop on formal methods in software practice*, pages 103–108, Clearwater Beach, Florida USA, March 1998.

[10] R. K. Keller, M. Cameron, R. N. Taylor, and D. B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.

[11] E. Madelaine and R. de Simone. *The FC2 Reference Manual*. Available by ftp from cma.cma.fr:pub/verif as file fc2refman.ps, INRIA.

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

[13] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 105–118, Melbourne, Australia, May 1992.

[14] W. J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV4)*, pages 49–59, Victoria, British Columbia, October 1991. ACM SIGSOFT, ACM Press.

[15] W. J. Yeh and M. Young. Re-designing tasking structure of Ada programs for analysis:A case study. *Software Testing, Verification, and Reliability*, 4:223–253, 1994.

[16] M. Young, R. Taylor, D. Levine, K. A. Nies, and D. Brodbeck. A concurrency analysis tool suite for Ada programs: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):65–106, Jan 1995.