Refactoring Design Models for Inductive Verification*

Yung-Pin Cheng
Dept, Info. and Comp. Education
National Taiwan Normal Univ.
162 Hoping E. Rd., Taipei 106
Taiwan
+886 2 23622841 ext 33
ypc@ice.ntnu.edu.tw

Abstract

Systems composed of many identical processes can sometimes be verified inductively using a network invariant, but systems whose component processes vary in some systematic way are not amenable to direct application of that method. We describe how variations in behavior can be "factored out" into additional processes, thus enabling induction over the number of processes. The process is semi-automatic: The designer must choose from among a set of idiomatic transformations, but each transformation is applied and checked automatically.

Keywords

Refactoring, Network Invariants, Parameterized System, Compositional Analysis, Concurrency

1 Introduction

When applying finite-state verification methods to a system with many identical processes, one would prefer to perform an inductive verification that applies to arbitrary size instances of the system. A popular approach to verifying systems parameterized by size is to construct a so-called network invariant and then check if the invariants pass the test of an induction framework (explained in section 2) such as those used by Wolper [24] and Kurshan [14].

*Effort supervised by Michal Young, Dept. of Comp. and Info. Sciences, University of Oregon and sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0034. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

The induction framework, however, assumes the behaviors of a process are constant, i.e., the finite-state machines representing the behaviors are fixed – with constant transition relations and number of states. This assumption could be true for some hardware systems, some protocols in which processes communicate by a shared bus, or linearly structured systems in which processes only communicate with its right or left neighbor. However, in many other application domains, one or more of the individual processes varies in some systematic way depending on the size of the system.

Standard induction frameworks cannot be directly applied to systems in which individual process behaviors are parameterized by system size. We describe how models of such systems can be transformed by *refactoring* to make induction applicable. The transformations "factor out" the variations in behavior into additional processes. Each refactoring step maintains equivalence between the original processes and compositions of the factor processes, so that the final refactored model is equivalent (weakly bisimilar) to the original model. Refactoring can be useful for improving the complexity of compositional state-space analysis in general [4], and is particularly useful for enabling inductive analysis.

This paper is organized as follows: We review the inductive finite-state verification in Section 2. In section 3, we describe the refactoring technique and its application to an example system. In section 4, we illustrate how the refactored example system can be verified inductively. Section 5 is a discussion. Section 6 and section 7 end the article with related work and conclusions.

2 Induction

Typically, to apply automatic finite-state verification techniques, a system is modeled as several finite-state machines

which communicate among themselves or with their environment. A feasible model for verification is one that has constant number of finite-state machines and constant transition relation and number of states in each finite-state machine. However, in practice, systems may be parameterized by size. They can be parameterized by the number of identical components, the number of data values (e.g., the length of a bounded buffer), or the number of control commands (e.g., a protocol that allows retransmission of messages over a lossy channel at most n times). Let a system S of size i be denoted as S_i and let all S_i form a family $F = \{S_i\}_{i=1}^{\infty}$. Let φ be some property of interest. The problem of verifying a parameterized system is to answer whether every S_i in F satisfies φ. This problem was shown to be undecidable in general [1].

Although the problem is undecidable in general, specific families may be solvable. A popular approach is using the induction framework described by Wolper [24] and Kurshan [14]. The framework can be generally described as follows: Consider the family F above and let I be some identical component. We assume $S_{i+1} = S_i||I$ for $i \geq 1$, where '||' is some composition operator. Let $\preceq$ be a preorder relation [12] over processes, and φ be a property of interest, such that

$$(P \preceq Q) \wedge (Q \models \varphi) \;\; \Rightarrow \;\; P \models \varphi.$$

If we can find a process Inv, such that

$$Inv \models \varphi \tag{1}$$

$$S_1 \preceq Inv \tag{2}$$

$$Inv \parallel I \preceq Inv, \tag{3}$$

we can use the fact that parallel composition is monotonic with respect to the preorder to infer from (3) that

$$\begin{gathered} Inv \parallel I \preceq Inv \\ (Inv||I)||I \preceq Inv \\ \vdots \\ Inv||I|| \cdots ||I \preceq Inv. \end{gathered} \tag{4}$$

Finally, we infer from (2) and (4) that

$$S_1||I|| \cdots ||I \preceq Inv. \tag{5}$$

Hence, using and the assumption $S_{i+1} = S_i||I$ and (1), we conclude that every S_i in F satisfies φ. A process Inv satisfying the above induction step is called *a network invariant*. In general, a network invariant may not exist due to the undecidability.

Parameterization, however, can affect models in two ways:

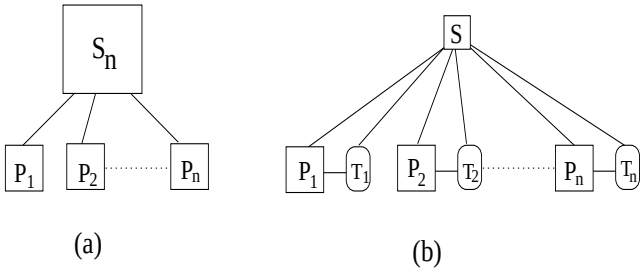


Figure 1: (a) The system G_n before refactoring. (b) The system G_n after refactoring.

1. Models add/remove processes when the system size is increased/decreased.

2. The behaviors (transition relation and number of states) of processes grow/shrink when the system size is increased/decreased.

In the literature, systems shown to pass induction framework mostly vary by size in first way. However, there are many systems whose parameterization affects models in second way or both. The induction framework used by Wolper and Kurshan can not be directly applied to systems with parameterized behaviors unless it happens that those variations are consistent with the preorder relation. When $\parallel$ is parallel composition, and $\preceq$ is one of the usual order relations in process algebra, this will not generally be the case.

Consider, for example, a system consisting of one control process and many identical slave processes. Let the system be $G_n = S_n \parallel P_1 \ldots \parallel P_n$, where S_n's behaviors are parameterized by n and processes P_i communicates with S_n by a private channel indexed with i. In Fig. 1(a), we show the example's communication structure, where a solid line represents a communication with a private channel between processes. While increasing n by 1, G_n becomes $G_{n+1} = S_{n+1} \parallel P_1 \ldots \parallel P_n \parallel P_{n+1}$. Note that $S_n \neq S_{n+1}$, so $G_{n+1} \neq G_n \parallel P_{n+1}$. G_{n+1} now makes a cycle through $n+1$ interactions over $n+1$ private channels, rather than n.

Suppose we found a putative invariant Inv for the family $\{G_i\}_{i=1}^{\infty}$. To complete the induction, we would next need to show that $Inv \parallel P_{n+1} \preceq Inv$. Recall that if this holds, we can infer from it with (2) to obtain (5). But, from (5) to the conclusion that every S_i satisfies φ, we need the assumption $G_{i+1} = G_i||I$. This is where the induction framework will usually fail for parameterized systems like $\{G_i\}_{i=1}^{\infty}$.

What we can do to fit such a system into the induction framework is "factor" the single process S_n (as in Fig. 1) into a much smaller fixed process S, independent of n, and

several identical $T_i, i = 1 \text{ to } n$, with transition labels renamed according to i. Each T_i communicates with the corresponding process P_i. This refactoring can be done in a way that maintains behavioral equivalence, i.e., the behavioral equivalence relation, S_n is weakly bisimilar to $(S \parallel T_1 \parallel \ldots \parallel T_n)$. (The next section describes how such refactoring can be partly automated.) So, after refactoring,

$$G_n = (S \parallel T_1 \parallel \ldots \parallel T_n) \parallel P_1 \parallel \ldots \parallel P_n.$$

If $S_{n+1} = (S \parallel T_1 \parallel \ldots \parallel T_n \parallel T_{n+1})$, an increase of n by 1 would result in a system

$$G_{n+1} = (S \parallel T_1 \parallel \ldots \parallel T_n \parallel T_{n+1}) \parallel P_1 \parallel \ldots \parallel P_n \parallel P_{n+1},$$

Note that, when n is increased by one, a pair of processes $\{T_{n+1}, P_{n+1}\}$ is added in the new structure.

Using the new structure of G_n, we can apply the induction framework to verify a system with an arbitrary number of P_i larger than n. If we transform safety properties into a deadlock detection problem (see [5, 25]), then we can use refactoring to inductively verify safety properties.

3 Refactoring

Since finding a network invariant is in general undecidable, no one is able to find a fully automated solution that is guaranteed to work for arbitrary systems. A conventional wisdom is that the network invariant can be refined from a base system (e.g., S_1 in family $\{S_i\}_{i=1}^{\infty}$), but the refinement is a creative endeavor which usually requires manual intervention and is hard to automate. Likewise, refactoring models to enable induction is a creative task that requires experience and an understanding of the the system to be analyzed. The designer must choose from among a set of idiomatic transformations and apply them in a right order. Nonetheless, tool support is possible for bookkeeping, helping the user recognize and apply potential transformations, and verifying that each step is sound. Although tool support cannot completely shield the user from understanding the finite-state model, it can hide many details of the transformations and checking.

Fig. 2 and 3 show the views of a refactoring operation provided by a prototype refactoring tool we have constructed. The topological view in Fig. 2 shows the overall communication structure of the model being refactored, as represented by a description in the ACME interchange format for software architecture descriptions [9]. Some operations, like grouping processes into subsystems and rearranging the hierarchy of scopes, can be performed directly at the topological level, with renamings and other simple transformations on the underlying processes being performed automatically.

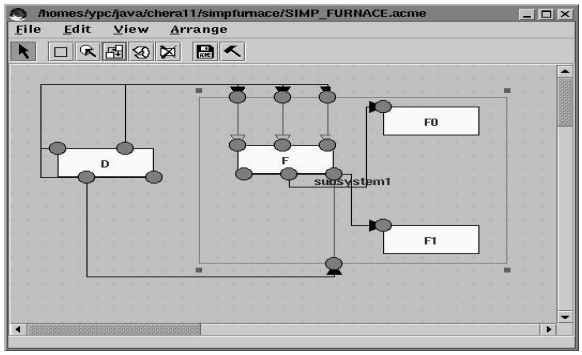


Figure 2: Topological view of an example using the prototype tool.

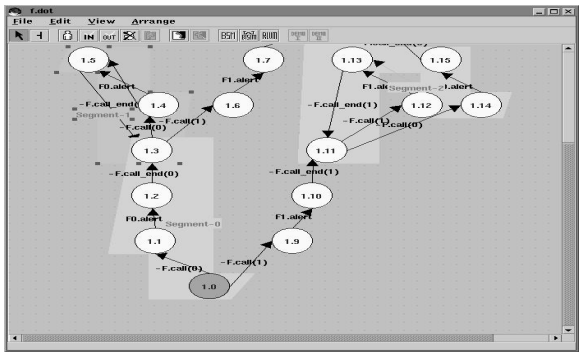


Figure 3: A view of the process graph of a process, which is about to be decomposed. Highlighted regions of the process graph have been selected for moving to a new process.

Decomposing a single process requires more complex manipulations of its process graph[1] which require additional guidance from the user. We have built a tool which provides process-graph view of an individual process (Fig. 3) and allows the designer to select portions of the process graph to be removed and incorporated in a new process. The newly created process will result in a structure modification that will be shown in the window of topological view.

Identifying parts of the process graph currently is a manual operation, but the main transformation is automated. Note that the size of the process graph that must be manipulated by the user does *not* depend on the size of the overall system, since it represents only a single component process. Prospects for further automation are discussed in section 7.

[1] More precisely, these process graphs are labeled transition systems with CCS semantics [16, 17].

3.1 Refactoring a remote temperature sensor system

In this section, a remote temperature sensor system (RTSS) (described originally by Sanden [21] and adapted by Yeh and Young [26]) is refactored and is verified (see section 4) by the induction framework. The remote temperature sensor system is a software-driven system that periodically reports the temperatures of an array of furnace devices. The system is parameterized by the arbitrary number of furnaces. Let the number of furnaces be fn and the furnaces be indexed from 0 to $fn - 1$. The overall structure of the system is shown in Fig. 4. The furnaces, shown on top-right portion of the figure, are located at a remote site. Requests sent from client to the sensor system use a simple ACK/NAK protocol. An example run is as follows: Initially, task UI sends a request (containing a furnace index) for each furnace. The request is packed into a control packet and delivered over a lossy channel. To ensure the packet is transmitted reliably, alternating bit protocol is used, which is implemented by the five tasks in CP subsystem. The five tasks are located either at client side or remote site. Once the packet arrives module FPACK, the task INTR alerts the furnace specified in the control packet. The alert may be lost but must get through at first time as an initialization. Once a furnace is initialized by first alert, it reads the temperature from THERMOMETER periodically or whenever an alert is received. After reading the temperature, it sends the temperature and its furnace index back to UI by data packets over alternating bit protocol again (see DP). While fn is small, the system can be composed compositionally by a hierarchy (UI (CP CP_INPUT) (FPACK THERMOMETER) (DP DP_INPUT)).

The general strategy of refactoring is to identify any processes whose structure depends on the size of the overall system and to refactor each of those processes into an invariant part and one or more variant parts, where such that the variation is in the number of variant parts rather than their structure. In this case, INTR, TINTR, tasks in CP_MODULE and tasks in DP_MODULE differ depending on the size of the system, so each of those processes must be refactored.

In principle, every process with behavior differing by system size must be refactored. However, for systems which are well-structured, like the furnace system described here, their modules may yield simple external behaviors. For example, CP_MODULE (which is obtained by composing (CP CP_INPUT) compositionally) though contains 6 tasks, its external behavior is simple and regular with respect to system size. Fig. 5 shows the external behavior of CP_MODULE with 2 furnaces. In this case, instead of refactoring the 6 original tasks of CP_MODULE, we choose to refactor CP_MODULE's external behaviors. This action can be easily justified by that our refactoring is equivalence preserving (explained later). The simplicity of CP_MODULE's interface behaviors makes a good example for describing refactoring. Thus, we skip the tedious steps of refactoring 6 original tasks and show how CP_MODULE's interface behavior is refactored.

To avoid any confusion, please note that in practice the external behavior of a module may not be simple, regular, and manageable. If that is the case, the module's external behavior is not suitable for refactoring (since refactoring is a partially automated process). A module which yields no simple interface behaviors often results from bad structuring, i.e., task interactions in the module cannot be effectively hidden and minimized. In this case, refactoring must operate on original tasks. Next, we break the original structure and find a better structure to group tasks into modules (recall that tool support for the restructuring is shown in Fig. 2). On the other hand, if a module yields simple interface behavior, it can save us the effort of refactoring at original tasks. Consequently, complexity of refactoring complicated systems can be reduced in this way.

3.1.1 The refactoring of CP_MODULE

In Fig. 5, the process graph inside the box is of CCS semantics. The label name *-send(0)* is a result of symbolic expansion of an Ada statement *accept send(i)*, where i=0,1. It should not be understood as a value is passed by the edge. Therefore, for every value of i, a sequence of actions *-send(i).call(i).callend* is added to the process graph.

We label an edge with prefix '–' to mean the action is at callee (or server) side. The edge *call_end* models the *do* block of an *accept* statement. The *accept call(i)* statement in INTR has a *do* block. So, whenever CP_MODULE issues *call(i)*, it must wait for the *do* block in INTR to complete, which is then modeled by *call_end* in CP_MODULE and *-call_end* at the end of *do* block. The box and ports are to illustrate its interfaces to task UI and task INTR.

The overall goal of refactoring is to recognize and separate parts of a process that have essentially been duplicated for dealing with different parts of the system. There is control structure in the original program that is parameterized by processes that a particular process or module is communicating with. This control structure by symbolic expansion becomes variation in the structure of the process or module. That is the kind of variation that can be removed, if we can transform it into extra processes instead. For example, one simple goal of refactoring for induction is to make

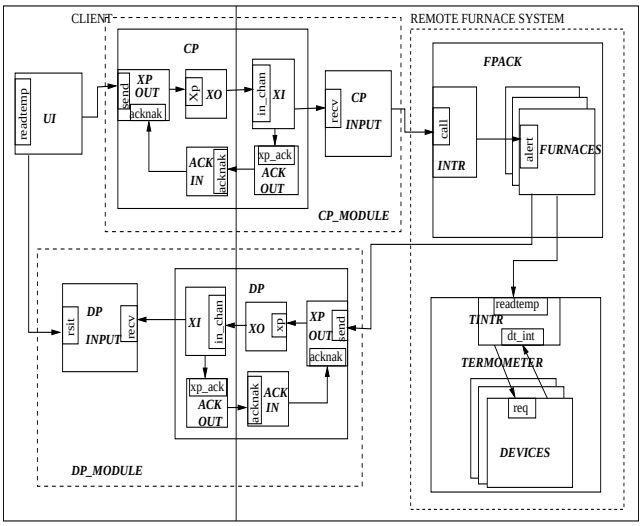


Figure 4: The overall structure of the remote temperature sensor system.

CP_MODULE independent of fn. To accomplish the goal, we need four subsequent transformations below:

Transformation I: Edge relabeling

The first transformation for refactoring CP_MODULE is to help recognition of the variant parts which essentially deal with different furnaces. In CP_MODULE, *-send(0)* and *call(0)* can be easily classified as linked to furnace 0, *-send(1)* and *call(1)* can be easily classified as linked to furnace 1, but *call_end* can not. In this example, we intend to classify *call_end* into either linked to furnace 0 or linked to furnace 1 so that CP_MODULE can become a clean, simple task (later shown in Fig. 9) and irrelevant to fn. The intended classification involves renaming *call_end* to either *call_end(0)* or *call_end(1)*. However, such kind of relabeling is not supported by the relabel operator (a.k.a. [a/b]) in CCS. We need a relabeling that is less strict.

Recall that *call_end* is an artifact of modeling Ada's *accept/do* semantics. So, a *call_end* is paired to a particular *call(i)*. Using this as guidance, we can relabel CP_MODULE into Fig. 6. Since action names in CCS are in pairs, we need to rename *-call_end* in INTR as well. Consequently, the interface between CP_MODULE and INTR is changed.

To justify the relabeling, we introduce a notion of equivalence. Since CP_MODULE and INTR are modified by this relabeling, CP_MODULE and INTR are viewed as a subsystem. The equivalence we propose is that the subsystem's behavior remains equivalent (weakly bisimilar) before and after the transformation. It can be expressed by the following equation:

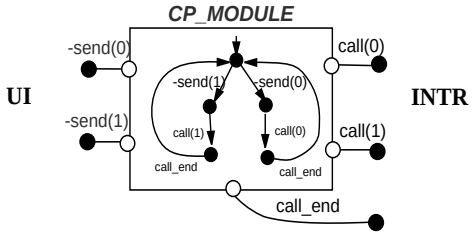


Figure 5: The module behavior of CP_MODULE.

$$(CP\_MODULE||INTR) \setminus \{call\_end\} \approx$$

$$(CP\_MODULE||INTR) \setminus \{call\_end(0), call\_end(1)\},$$

where '$\approx$' is the weak bisimulation of CCS and '$\setminus$' is the restriction operator of CCS. Verifying the equation can assure the correctness of our relabeling strategy.

The relabeling above seems ad hoc and lack of generality. In general, whether a set of edges can be relabeled safely (according to our notion of equivalence) can be determined easily. Using the above case as an example, a simple algorithm is:

```
1.Mark every edge call_end with
  a distinct and unique number.
2.Mark every edge -call_end with
  a distinct and unique number
3.Perform (CP_MODULE||INTR) and monitor
 the rendezvous during composition.
   If (an edge call_end rendezvous
    with an edge -call_end) then
      merge their number into a set
    or
       merge the sets to which their
       number already belong into a
       bigger set.
 end if.
```

The algorithm may partition all the numbers into disjoint sets. Let the marked edges be partitioned like their associated numbers. Thus, the edges in a disjoint set will not rendezvous with edges in other sets. So, trivially, relabeling edges of a disjoint set altogether is safe. Once we know which set of edges can be relabeled safely, it is then up to the designers to determine what relabeling is needed.

Transformation II: 1st Behavior decomposition

The second transformation is to decompose CP_MODULE by extracting away the behavior linked to furnace 0. The extracted behavior is then wrapped into a new process. With

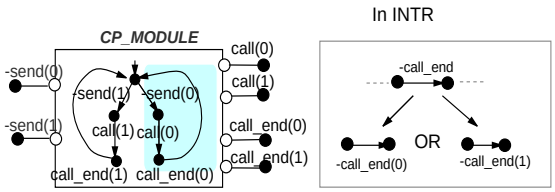


Figure 6: CP_MODULE after edge relabeling transformation. In this transformation, *call_end* is renamed as *call_end(0)* or *call_end(1)* in INTR and CP_MODULE.

the new process introduced into the system, the synchronization structure of the system is changed, thus, enabling the induction framework or compositional analysis that is doomed by bad as-built structure.

The first step of behavior decomposition is to identify the behavior to be extracted. In Fig. 6, we use a shaded area to mark the behavior to be removed. Usually, this is where human assistance is needed. After segments of behavior have been properly identified and marked, the rest of the transformation is automatic. Functions of the transformation include purging the marked behaviors from CP_MODULE, wrapping the marked behaviors into a new task, and inserting new communications to preserve equivalence. Fig. 7 illustrates the result of this transformation. The new task created by this transformation is CP_SUB0. Edge labels highlighted by grey background are the new communications inserted by the transformation. In UI, every edge labeled *send(0)* is replaced by two edges, labeled as *get(0)* and *send(0)*.

In Fig. 7, *-send(0)* is redirected to CP_SUB0 but *-send(1)* still remains in CP_MODULE. If nothing is done to constrain the two tasks, it is possible for *-send(0)* and *-send(1)* to accept rendezvous before the other one completes its service —- which is not equivalence preserving. To remedy the problem, in UI, every *send(0)* is guarded by *get(0)*, in CP_MODULE, *-get(0)* and *-release(0)* (which behave like a semaphore of value 1) are inserted to fill the vacant areas preoccupied by the marked behaviors, and in CP_SUB0, *release(0)* is appended at the end of marked behavior. So, whenever a *-send(0)* is called, the caller must acquire *-get(0)* first to lock CP_MODULE. Once CP_MODULE is locked, attempts to rendezvous with *-send(1)* would be impossible. Next, *-send(0)* is redirected to CP_SUB0 to complete the marked behavior. Finally, after CP_SUB0 completes the marked behavior, CP_SUB0 releases CP_MODULE by invoking *release(0)*.

How equivalence is preserved by this transformation? The same notion of equivalence in transformation I is applied. Here, UI, CP_MODULE are modified. So, (UI CP_MODULE)

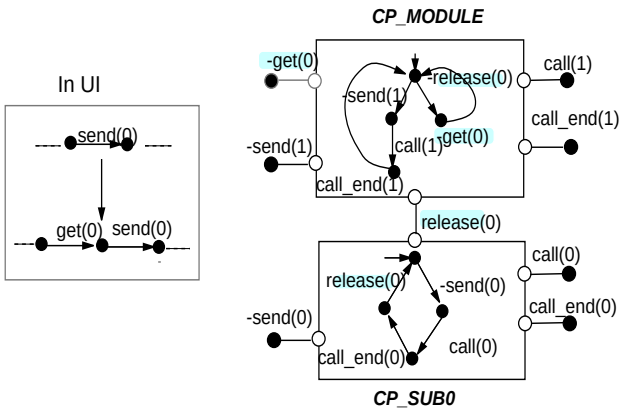


Figure 7: CP_MODULE after Transformation II. CP_SUB0 is the new process created by the transformation. *-send(0)*, *call(0)*, *call_end(0)* are all redirected to CP_SUB0. Every *send(0)* in UI is guarded by new communication *get(0)*.

is viewed as a subsystem. In this view, *send(0)* and *send(1)* are internal and restricted. Similarly, after refactoring, we view (UI CP_MODULE CP_SUB0) as a subsystem. In this view, *send(0)*,*send(1)*,*get(0)*, and *release(0)* are internal and restricted. The following equation can be verified to justify the transformation:

$$(UI||CP\_MODULE) \setminus \{send(0), send(1)\} \approx$$

$$(UI||CP\_MODULE||CP\_SUB0)\setminus\{send(0), send(1), get(0), release(0)\}.$$

Transformation III: 2nd behavior decomposition

The third transformation is the same as transformation II, only the marked behavior is those linked to furnace 1. The result of this transformation is shown in Fig. 8, in which CP_MODULE behaves as a pure semaphore. To this stage, all the behaviors linked to furnaces are redirected to CP_SUB0 or CP_SUB1 respectively.

Transformation IV: Semaphore simplification

Despite the simple behavior, CP_MODULE in Fig. 8 is still parameterized by fn. To make it independent of fn, we need the forth transformation. But before that, let's review CCS's rendezvous semantics. In CCS, if there are two processes both ready to communicate by action a but there is only one process ready to communicate by co-action $\bar{a}$, the first two processes compete for the rendezvous. This is known as two-way rendezvous, as opposed to multi-way rendezvous of CSP. This important characteristic allows us to further simplify the semaphore in Fig. 8. Result of the

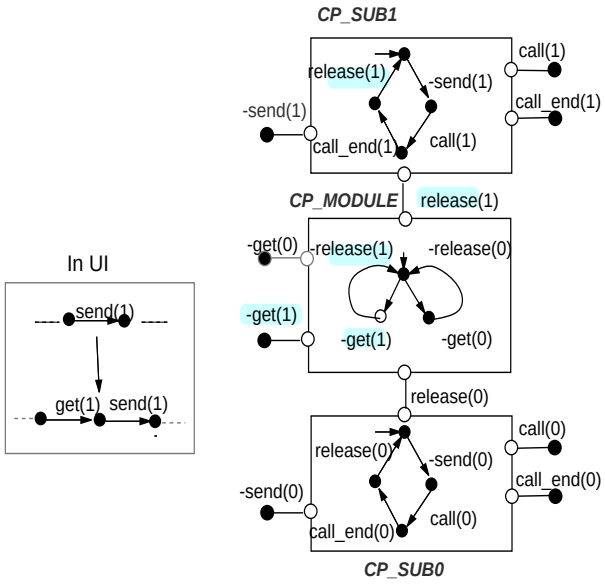


Figure 8: CP_MODULE after Transformation III.

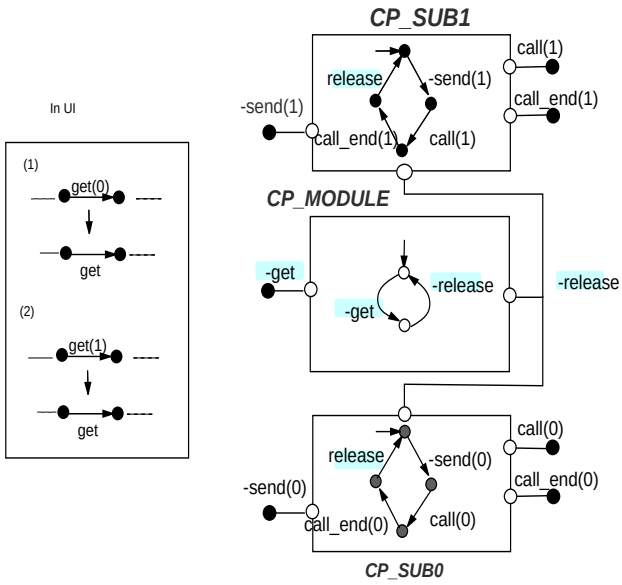


Figure 9: CP_MODULE after Transformation IV, where *get(0)* and *get(1)* are merged into one *get.* So are *release(0)* and *release(1)*.

simplification is shown in Fig. 9, where *get(i)* and *release(i)* are all renamed to new names *get* and *release* respectively. CP_MODULE now only have two ports but each port may have more than one process attached. For example, port *release* is now attached by CP_SUB0 and CP_SUB1. Using the same notion of equivalence as before, the following equation can be verified to justify this simplification:

$$(UI||CP\_MODULE) \setminus \{get(0), get(1), release(0), release(1)\} \approx$$
$$(UI||CP\_MODULE) \setminus \{get, release\}$$

The final structure of CP_MODULE

At last, CP_MODULE in Fig. 9 holds a structure amenable to the induction framework. First, CP_MODULE is independent of fn; that is, its process graph no longer varies by fn. Second, this structure is meant to be extended easily: While fn is increased, we simply add another identical process CP_SUB2 and attach its *release* port to that of CP_MODULE. These two properties enable induction framework. Nonetheless, not all the process graphs can be refactored to have the two properties since verification of parameterized system is undecidable in general.

3.1.2 The refactoring of INTR

In remote temperature sensor system, the greatest challenge to refactoring is INTR's process graph. The reason why INTR is worth noting is that its process graph grows exponentially as fn increases. In this subsection, we show that INTR can still be refactored to hold the above two elegant properties.

The abstract behavior of INTR with 2 furnaces is illustrated below (written in an Ada-like language called PAL[25]):

```
task INTR is
furnace_id_t := 2 ;
fFirst: array[furnace_id_t] of boolean;
entry call(i: furnace_id_t);
begin
   for fid in furnace_id_t loop
      fFirst[i] := TRUE;
   end loop;
   loop
     accept call(i) do
       if fFirst[i] = TRUE then
            furnace[i].Alert;
            fFirst[i] := FALSE;
       else
          select
            furnace[i].alert;
          else
            null; // time out
          end select;
       end if;
     end call;
   end loop;
end INTR;
```

As shown by the code, when INTR accepts *-call(i)* from CP_MODULE, INTR alerts the *i*th furnace. The alert can be lost but not the first time. Otherwise, a deadlock would be presented as an artifact of incomplete modeling of task activation. So, a boolean array fFirst[] is added to remedy

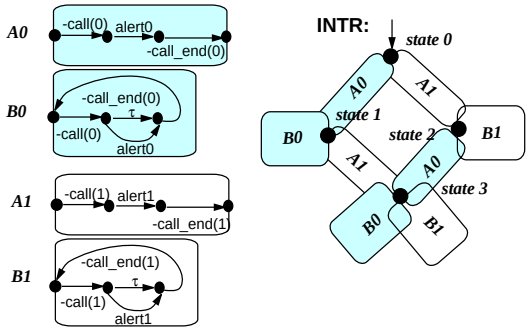


Figure 10: The process graph of INTR.

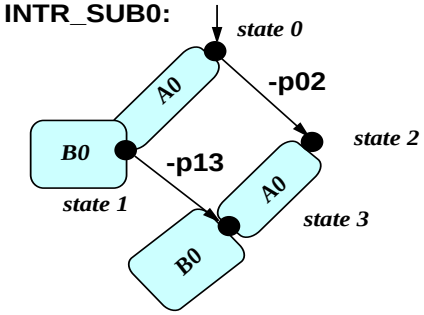


Figure 11: The new process with *patches* inserted.

this situation. However, the remedy causes process graph of INTR to grow exponentially because INTR must bookkeep the alerted condition of each furnace, which induces 2^{fn} of combinations.

The process graph of INTR of 2 furnaces is shown in Fig. 10. At the left, we use $A0$, $B0$, $A1$, and $B1$ to represent segments of behaviors, where τ is to emulate timeout (caused by loss of alert). At the right is the process graph of INTR constituted by these segments. There are four major states in the graph. At each major state, INTR can accept *-call(0)* or *-call(1)*. If *-call(0)* is accepted, the control goes to $A0$ (if the furnace has never been alerted) or $B0$ (if the furnace is alerted already). For example, state 0 means no furnaces have been alerted before. State 1 means furnace 0 has been alerted at least once, but furnace 1 has not. So, with 2 furnaces, there are 4 major states; with 3 furnaces, there are 8 major states; and so forth.

To decompose INTR, the same transformation in transformation *II* & *III* is used, only input is more complicated. This time, the behavior linked to furnace 0 is distributed over four segments ($A0s$ and $B0s$). They should be extracted away and then wrapped into a new task all together. So, after the segments with behaviors linked to furnace 0 are identified and marked, the transformation purges the marked segments from INTR. Next, *-get(0)* and *-release(0)* are inserted to fill every vacant area the marked segments left. Finally, wrapping the segments into a new process is tricky because segments can be disconnected or not reachable from initial state (see Fig. 11). A new type of communication called *patch* may be inserted to reconnect the major states and preserve equivalence. *Patch* is meant to coordinate the refactored INTR and the new process. Image the refactored INTR and the new process are up and running. When refactored INTR changes it current state from 1 to 3 (via $A1$), the new process must do so immediately so that when *-call(0)* is redirected to the new process, the right segment is used.

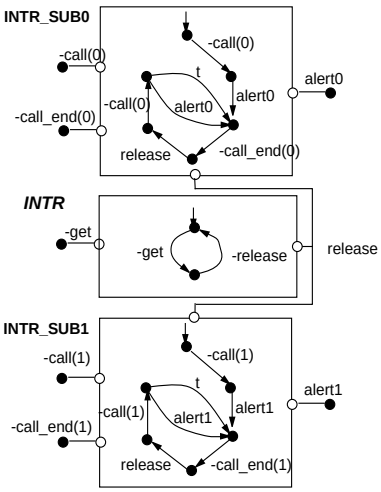


Figure 12: The refactored INTR.

So, a *patch* will be inserted between state 1 and state 3 in refactored INTR and the new process as a coordinator. Fig. 11 shows the result of inserting patches *-p02* and *-p13* into the new process. Accordingly, in INTR, a *p02* is inserted between segment *A1* and state 2 and a *p13* is inserted between segment *A1* and state 3.

The decomposition transformation, however, does not consider the repeated behavior patterns while inserting *patches*. For instance, in the new process, $A0$ and $B0$ repeat more than once. It is not always necessary to switch between major states if the redirected behaviors are the same. For example, at state 1 and state 3, the redirected behaviors are the same $B0$. A variant of weak bisimulation minimization[2] is used to minimize the new process. The variant considers *patches* differently so that repeated patterns can be minimized and some *patches* can be reduced. In the INTR case,

[2] Pure weak bisimulation minimization does not work because it treats the patches as normal actions.

Task name	Names of tasks after refactoring
CP_MODULE	***CP_MODULE***,CP_SUB0, CP_SUB1
INTR	***INTR***,INTR_SUB0, INTR_SUB1
TINTR	***TINTR***,TINTR_SUB0,TINTR_SUB1
DP_MODULE	***DP_MODULE***,DP_SUB0,DP_SUB1
UI	***UI***,UI_SUB0, UI_SUB1
Device[i]	(not refactored)
Furnace[i]	(not refactored)

Table 1: The task names of refactored RTSS.

both patches in Fig. 11 are reduced. The same notion of equivalence-preserving is used to justify the minimization. The final result of refactored INTR is shown in Fig. 12.

4 The induction of RTSS

The tasks of refactored RTSS are summarized in Table 1. In second column, task names in bold-italic font are semaphore tasks which are independent of fn. Let

$$C = CP\_MODULE||INTR||TINTR||DP\_MODULE||UI,$$

$$F_i = CP\_SUBi||INTR\_SUBi||TINTR\_SUBi||DP\_SUBi||UI\_SUBi,$$

and

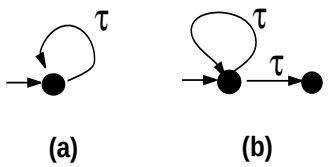
$$R_i = Device[i]||Furnace[i].$$

Let $RTSS(1)$ be the system with 1 furnace (starting from index 0). We have

$$RTSS(1) = C||F_0||R_0.$$

Verification of a safety property is translated into a deadlock detection problem [5, 25]. The places to embed inductive safety properties (which holds for arbitrary number of furnaces) are F_i and R_i for all i. If no safety property is embedded, the verification problem is equal to check if $RTSS$ is free of deadlock with respect to arbitrary number of furnaces.

Let $RTSS^*(1)$ denote the $RTSS(1)$ embedded with safety properties. The base step of the induction is to verify whether $RTSS^*(1)$ is deadlock free. Note that compositional techniques are used in the verification to mitigate state explosion. To detect if $RTSS^*(1)$ has deadlock, one simple way is restricting all the communications, i.e., making them not observable. If the final process graph is akin to figure (a) below, then $RTSS(1)$ satisfies the property. On the other hand, if a deadlock state presents in the final process graph like figure (b) below, $RTSS(1)$ violates the property.



To pass the induction step, we need a network invariant. The $RTSS^*(1)$ without exporting any communications, however, is infeasible for being an invariant. Recall that ports *-get* and *-release* of every task in C are meant to be attached by additional identical processes. These communications must be exported so that $RTSS$ can be extended to arbitrary size. Let Inv be $RTSS^*(1)$ with *-get* and *-release* of semaphore tasks being exported. The induction step is to verify:

$$(Inv||F_i||R_i) \approx Inv.$$

If the equation holds, we conclude the safety property is satisfied with respect to arbitrary number of furnaces. Our experiment shows that Inv is an effective network invariant for $RTSS$.

5 Discussion

Besides the remote temperature sensor system, some systems in the literature are also refactored to see if they can pass the induction verification. The elevator system [20] is another example worth mentioning. Most of the behavior of elevator system can be refactored as expected but some behaviors are not. They are for task initialization and termination. Their patterns are a sequence of commands which are issued to elevators one after another. We have not yet managed to find a way to refactor the patterns into a structure suitable for induction framework. We choose to ignore the problematic behaviors (since they are only a minor part of elevator's behaviors) and focus on the continuously running parts. Thus, a network invariant can be constructed and the induction framework becomes applicable.

6 Related Work

In contrast to network-invariant based approaches, for some restricted system topologies, whether a parameterized system satisfies a (restricted) temporal specification has been shown to be decidable. So, if the domain is restricted to a particular topology, the problem can be approached by projecting infinite state space into finite state space, verifying the given properties on finite state space, and then concluding the properties hold in the infinite state space as well. In

the late 80s, non-automatic approaches were proposed by Clarke et al [3, 6]. Later, systems comprising a single control process with arbitrary number of identical processes are studied by German [10] and Emerson [8], both providing (semi) automatic methods. Other topology with processes communicating in a ring structure is studied in [7]. They show there exists a k such that the correctness of a ring structure of size k implies the correctness of networks of all sizes.

Decidability of verifying parameterized systems in which processes communicate by private channels is studied by Girkar and Moll [11]. They show the problem is undecidable in general, but assuming a particular topology with a synchronizer and arbitrary number of user processes, the deadlock detection problem is decidable.

In invariant-based approaches, finding the network invariant often requires human ingenuity and trial-and-error. Nevertheless, under some particular topology and conditions, automatic computation for the network invariants is possible [22, 18, 19, 2]. An attempt to generalize automatic computation of network invariants is made by Clarke et al [15]. They use context-free network grammar to describe the topology of networks and provide a heuristic method to find the invariant. The procedure is not guaranteed to find the invariant and might not terminate, which is consistent with the decidability result. Clarke et al. also introduce a new way of specifying properties for the invariant-based approaches, making specifying properties closer to the fashion of temporal logic.

A case study by Valmari and Kokkarinen [23] on lossy channels is, to our knowledge, the approach closest to that described here. Valmari and Kokkarinen studied a protocol with lossy channels in which the system allows messages to be retransmitted at most n times – a channel parameterized by n. The channel's behaviors resemble the initialization and shutdown sequences of the elevator example. They replace the single parameterized channel by the composition of n smaller processes called *counter cells* under CSP (multi-way rendezvous) semantics [13]. Valmari and Kokkarinen also adopt compositional analysis to compose the processes and result in showing a liveness property in this particular case.

Valmari and Kokkarinen's method of replacing the channel's behaviors with the composition of n processes captures the spirit of our refactoring, but is rather specialized to the particular case they considered. It is interesting to note that Valmari and Kokkarinen's approach would solve the problem that forced us to omit startup and shutdown sequences in our inductive analysis of the elevator system. Unfortunately, adopting CSP semantics to solve this problem would make it impossible to apply some of the other transformations that we used to refactor the elevator system.

7 Conclusions

Inductive verification using network invariants cannot be directly applied to systems in which individual component processes vary in some systematic way depending on the size of the system. We have described how models of such systems can be transformed — *refactored* — into equivalent models in which inductive verification can be applied.

Refactored models are composed in a modular and hierarchical manner to avoid state explosion during an inductive verification. Refactoring, and verification of the soundness of transformation steps, is performed locally so that its cost is not proportional to the size of the system.

References

[1] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, pages 207–309, 1986.

[2] F. Balarin and A. L. Sangiovanni-Vincentelli. On the automatic computation of network invariants. In *CAV94, LNCS 818*, pages 234–246, 1994.

[3] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81:13–31, 1989.

[4] Y.-P. Cheng and M. Young. Refactoring design models for compositional analysis and conformance testing. (in preparation).

[5] 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.

[6] E. M. Clarke and O. Grümberg. Avoiding the state explosion problem in temporal logic model checking algorithms. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 294–303, August 1987.

[7] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *22nd ACM Symposium on Principles of Programming Languages*, pages 85–94, 1995.

[8] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems. In

8th Conference on Computer Aided Verification, LNCS 1102, pages 87–98, 1996.

[9] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[10] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, July 1992.

[11] M. Girkar and R. Moll. New results on the analysis of concurrent systems with an indefinite number of processes. In *LNCS*, volume 836, pages 65–80, 1994.

[12] M. Hennessy. *Algebraic Theory of Processes*. MIT Press Series in the Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1988.

[13] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[14] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.

[15] E. M.Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. In *ACM Transactions on Programming Languages and Systems*, volume 19, pages 726–750, 1997.

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

[17] R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.

[18] J.-K. Rho and F. Somenzi. Inductive verification of iterative systems. In *Proceedings of the 29th Design Automation Conference*, pages 628–633, Anaheim, California, USA, June 1992.

[19] J.-K. Rho and F. Somenzi. Automatic generation of network invariants for the verification of iterative sequential systems. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings. Lecture Notes in Computer Science, Vol. 697, Springer, 1993*, pages 123–137, 1993.

[20] 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.

[21] B. Sanden. Entity-life modeling and structured analysis in real-time software design–a comparison. *Communications of the ACM*, 32(12):1458–1466, December 1989.

[22] A. P. Sistla. Parameterized verification of linear networks using automata as invariants. In *CAV '97, LNCS 1254*, pages 412–423, 1997.

[23] A. Valmari and I. Kokkarinen. Unbounded verification results by finite-state compositional techniques: 10^{any} states and beyond. In *International Conference on Application of Concurrency to System Design, Proceedings*, pages 75–85, Aizu-Wakamatsu, Fukushima, Japan, March 1998.

[24] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems, Volumne 407 Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, June 1989.

[25] 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.

[26] 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.