Compositional Verification by Model Checking for Counter-Examples

Tevfik Bultan
Department of Computer Science
University of Maryland
College Park, MD 20742

Jeffrey Fischer
Rational Software Corporation
8000 Westpark Drive
McLean, VA 22070

Richard Gerber
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Many concurrent systems are required to maintain certain safety and liveness properties. One emerging method of achieving confidence in such systems is to statically verify them using model checking*. In this approach an abstract, finite-state model of the system is constructed; then an automatic check is made to ensure that the requirements are satisfied by the model. In practice, however, this method is limited by the* state space explosion problem*.*

We have developed a compositional method that directly addresses this problem in the context of multi-tasking programs. Our solution depends on three key space-saving ingredients: (1) checking for counter-examples, which leads to simpler search algorithms; (2) automatic extraction of interfaces, which allows a refinement of the finite model – even before its communicating partners have been compiled; and (3) using propositional "strengthening assertions" for the sole purpose of reducing state space.

In this paper we present our compositional approach, and describe the software tools that support it.

1 Introduction

Computing systems are now used in applications like automobiles, aircraft, and assembly lines. These systems often have safety-critical requirements; the computation must *always* produce the right answer. The computations typically require the coordinated efforts of multiple concurrent tasks. Critical properties, such as mutual exclusion and freedom from deadlock, must be maintained by these tasks, but the temporal nature of these properties makes checking them a highly complex problem. This is particularly true when the properties depend on sequences of subtle task interactions. In such cases, some form of static analysis can often help a designer verify that the program will behave as expected.

Our general paradigm is this: to abstract our systems into mathematical models; to characterize specifications in terms of mathematical properties imposed on models; and to verify that models possess these properties. There is a wide range in capability and accuracy of various modeling techniques, as well as in the complexity of verifying properties of the models [7, 14, 16]. The particular technique we use is *model checking*.

Model checking refers to a family of algorithms used to *check* whether a state-transition structure satisfies a formula in some temporal logic. Several such logics [9, 13] and procedures [6, 10, 15] have been proposed for this use. Our work uses CTL [1] model checking [4], which runs in time linear in both the size of the structure and the length of the formula being verified.

The size of the structure normally dictates the cost of verification. When n models of size m are composed in parallel, the result can be a model of size m^n. This exponential blow up is commonly referred to as the *state space explosion problem*.

A considerable amount of research has been directed towards techniques for reducing the size of the model that gets constructed while applying model checking. Reduction techniques have generally followed one of two paths: (1) to avoid generating the entire structure, and instead to perform the check *compositionally* or *locally* [5, 8, 17, 18]; or (2) to form the whole structure, but encode it *symbolically* to reduce its size [2].

We have taken the compositional approach, albeit with several unique twists.

This work was supported in part by ONR grant N00014-94-10228, NSF grant CCR-9209333, and NSF Young Investigator Award CCR-9357850.

[1] Computation Tree Logic - a branching time temporal logic.

First, we negate a user-entered invariant, and then search for its counter-example. We show how this type of query leads to a sound and complete theory of compositional model-checking, the preliminary sketch of which was first reported in [11] (which in turn extended the hardware analysis work in [3]).

Second, we exploit Ada's rendezvous-style communication, and automatically extract "logical interfaces" for each task. This allows us to refine a task's model with respect to its communicating partners – even before *their* models have been generated.

Third, we show how simple, "strengthening assertions" can be used to reduce the cost of verification.

Finally, we have developed a Tcl/Tk-driven toolset, in which each phase of model-checking is implemented as a C++ class. In this manner, we have been able to measure the effects of various orderings of operations (e.g., compilation, composition, model-checking, etc.). We show some of these measurements in this paper.

The remainder of the paper is organized as follows. In Section 2 we motivate our compositional approach, and in Section 3 we give a more technical development of the theory, specially as it relates to verifying Ada programs. In Section 4 we show how our tool is used to support our approach, and in Section 5 we present some measurements of its effectiveness. We also overview some of the follow-on work we plan to pursue. We conclude in Section 6 with some remarks on the usefulness of this technology.

2 Overview of the Solution

Model checking requires two ingredients: a set of models (e.g., transition systems) and a set of properties expressed in temporal logic. Transition systems are generated from the source code of the program to be verified, each of which represents a task in the concurrent program. In our current implementation we concentrate on Ada programs. As the rest of this section shows, our method is general, and can be applied to any concurrent programming language.

The domain of the properties we are interested in includes two categories: safety and liveness. Safety properties express *invariants* of the system's behavior, i.e., they capture conditions that are expected to hold in each state of the system. A liveness property expresses a constraint on the future behavior of the system, i.e., that the current state of the system always *leads-to* an expected state.

After compiling the source code of the input program to the set of transition systems $T_i, T_j, \ldots$ a parallel composition, $T_i \| T_j \| \ldots$, can be formed, representing the concurrent executions of the set of transition systems. The problem of verifying that the concurrent program satisfies the given property is reduced to deciding whether or not the composite transition system satisfies F, where F is the temporal logic formula representing the given property. I.e., if $T_i \| T_j \| \ldots \models F$ then the property holds for the program.

Compositional approaches usually share a common objective – to solve n individual problems of size m, as opposed to one problem of size m^n. For several very basic reasons, this goal has generally eluded researchers in the area of automated finite-state analysis. Consider the mechanics of model checking, where one uses a function "$\mathbf{check}(T : \mathit{model}, F : \mathit{formula}) : \mathit{model}$," and expects it to return the sub-model of T satisfying the formula F. In most cases the property F is an "invariant" – which in CTL notation is written $\forall\Box G$ for some formula G. When $\forall\Box G$ is a requirement of a system T, we want G to hold in *all* of T's states.

But when T is the cartesian product of two models T_i and T_j, we have the following:

$$T_i \| T_j \quad \textbf{satisfies } \forall\Box G \quad \textbf{iff} \quad \mathbf{check}(T_i \| T_j, \forall\Box G) = T_i \| T_j.$$

And herein lies the problem with compositional analysis. Since T_i and T_j interact with each other, if we wish to determine whether $\forall\Box G$ holds, we will still end up generating the composite machine $T_i \| T_j$. When several concurrent models are involved, this technique can easily lead to "state space explosion."

Our approach *is* compositional, albeit at the expense of some generality. We restrict our input requirements to the common "global" ones; i.e., of the form $\forall\Box G$ ("in every state G holds"), $\forall\Diamond G$ ("every trace contains a state where G holds"), and $\forall \bigcirc G$ ("G holds in all next states"). Then we achieve compositionality by taking a formula-dependent approach to minimizing each model.

Our algorithm essentially turns a verification query into its dual, and it progressively shrinks each model based on the (negated) requirements formula (see Figure 1). For example, assume we wish to determine whether $\forall\Box G$ is true for $T_i \| T_j$. Then we have:

$$\mathbf{check}(T_i \| T_j, \forall\Box G) = T_i \| T_j \quad \textbf{iff} \quad \mathbf{check}(T_i \| T_j, \exists\Diamond\neg G) = \emptyset.$$

The formula "$\exists\Diamond\neg G$" means "in some reachable state $\neg G$ holds;" this is true if and only if G is not an invariant. Our

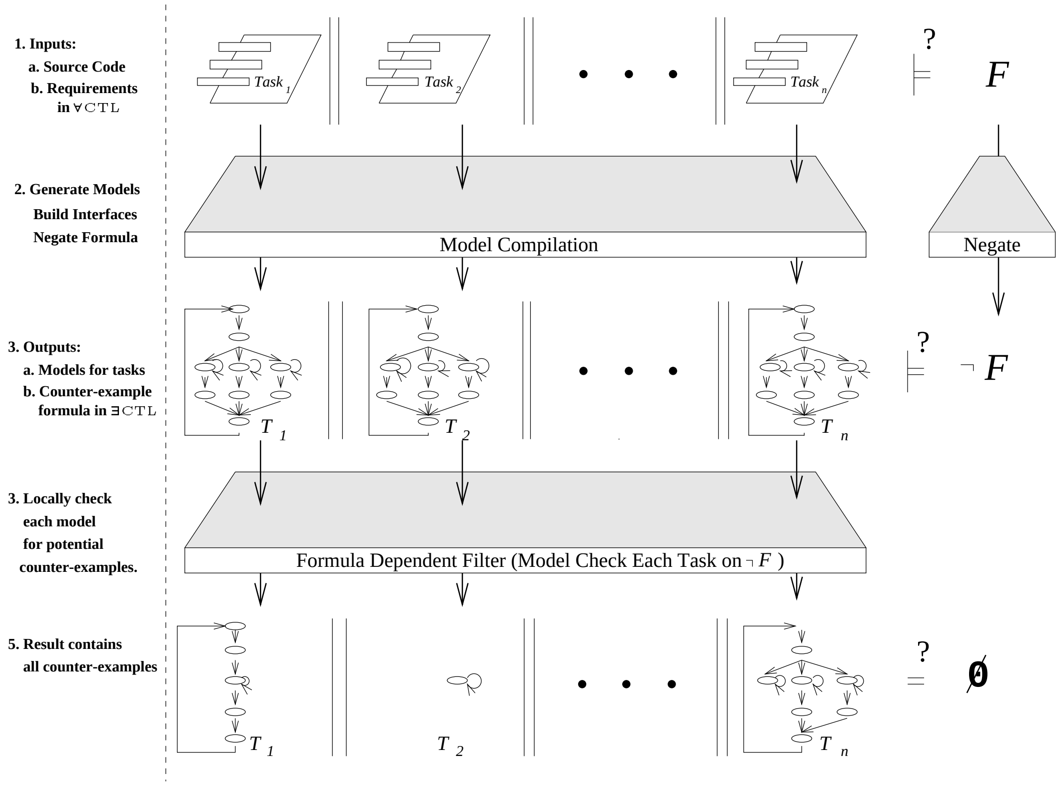

Figure 1: Our compositional verification method.

compositional checker **check** exploits simple facts like these, and hunts for "local" counter-examples to the input formula. Formally, the compositional result is:

$$\textbf{check}(\ T_i \| T_j, \exists\Diamond\neg G\ ) =$$
$$\textbf{check}(\ \textbf{check}(T_i, \exists\Diamond\neg G)\ \|\textbf{check}(T_j, \exists\Diamond\neg G)\ ,\ \exists\Diamond\neg G\ ).$$

The right-hand side will often be considerably less expensive, since the inner calls to **check** conservatively delete transitions known not to be involved in any counter-examples. Thus the final composite contains *all* potential counter-examples – but nothing else.

3 The Solution - Theory and Practice

In this section we describe the steps of our method in detail, by way of a running example.

3.1 An Example

Consider the Ada tasks shown in Figure 2(top), where a *program* is composed of n ($n \geq 3$) of these tasks. The $n-2$ middle tasks have a common structure (and differ only in their communications partners), while the 1st and nth are unique.

F	$::=$	$A \mid \forall\bigcirc F \mid \forall\Box F \mid \forall\Diamond F \mid F\,\forall\mathcal{U}\,F \mid F \wedge F \mid F \vee F$
A	$::=$	$\langle\text{atomic proposition}\rangle \mid \neg A \mid A \wedge A \mid A \vee A$

Table 1: ∀CTL.

As can be seen in Figure 2(bottom), we model an Ada program of n tasks as a product of n finite state machines. We abstract away data values and their influence on control flow. This allows us to concentrate on the task interaction, which is our chief concern – and it also keeps the associated decision problem tractable. Of course the price we pay may be a rather pessimistic analysis.

One requirement of our program may be the following property:

> *Whenever task $P_{i,i<n}$ has terminated, task P_{i+1} will eventually terminate.*

At the "user input level," this type of property is entered as a sentence in the universally quantified CTL (called ∀CTL), a grammar for which is presented in Table 1. In Table 2 we present the interpretation for the full CTL, i.e., the logic with both universal and existential quantifiers.

Using ∀CTL, and the annotated labels in Fig-

States	Task P_1	States	Task P_i	States	Task P_n
	task body P_1 is		task body P_i is		task body P_n is
	B : Boolean;	1	begin	1	begin
1	begin		p_i_outer: loop		p_n_outer: loop
	loop	5, 6, 7	accept Ready;	5, 6, 7	accept Ready;
5, 6, 7	P_2.Ready;	8, 9, 10	P_{i+1}.Ready;		p_n_inner: loop
	if B then		p_i_inner: loop	11	select
9, 10, 11	P_2.Quit;	14	select	13, 14, 15	accept Quit;
	exit;	16, 17, 18	accept Quit;		exit p_n_outer;
	end if;	19, 20, 21	P_{i+1}.Quit;		or
14, 15, 16	P_2.Cont;		exit p_i_outer;	17, 18, 19	accept Cont;
	end loop;		or		exit p_n_inner;
17	end P_1;	23, 24, 25	accept Cont;		end select;
		26, 27, 28	P_{i+1}.Cont;		end loop;
			exit p_i_inner;		end loop;
			end select;	21	end P_n;
			end loop;		
			end loop;		
		30	end P_i;		

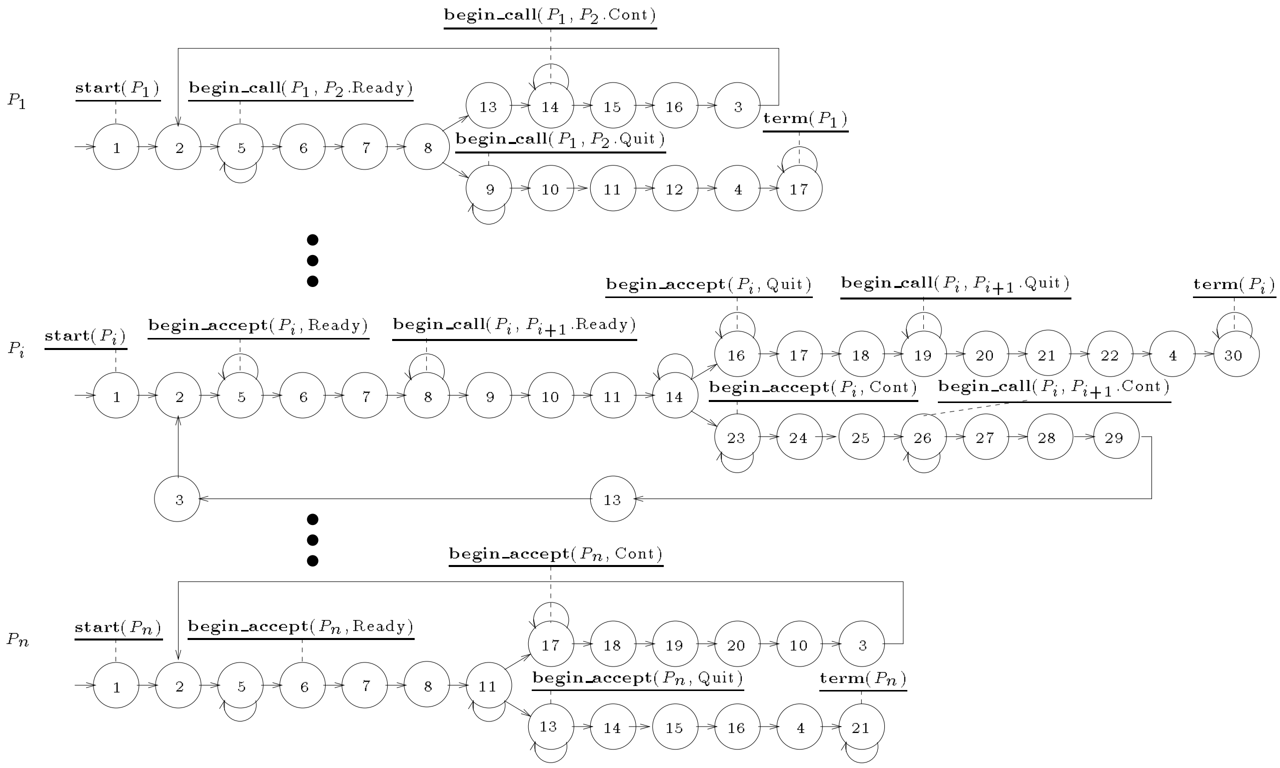

Figure 2: Source code (top), and transition relation graphs of n task chain example (bottom).

Our Notation	Interpretation
f	f is true in current state for atomic proposition f.
$\forall \bigcirc F$ ($\exists \bigcirc F$)	F is true on all (some) next states.
$\forall \Diamond F$ ($\exists \Diamond F$)	F is eventually true on all (some) paths.
$\forall \Box F$ ($\exists \Box F$)	F is invariantly true on all (some) paths.
$F \forall \mathcal{U} G$ ($F \exists \mathcal{U} G$)	On all (some) paths G holds at some state and F holds in every state on the path prior to that state.

Table 2: CTL temporal operators.

$\neg \forall \Box F$	$\equiv$	$\mathbf{True}\ \exists \mathcal{U}\ \neg F$
$\neg \forall \Diamond F$	$\equiv$	$\exists \Box \neg F$
$\neg \forall \bigcirc F$	$\equiv$	$\exists \bigcirc \neg F$
$\neg (F_1\ \forall \mathcal{U}\ F_2)$	$\equiv$	$(\exists \Box \neg F_2) \vee (\neg F_2\ \exists \mathcal{U}\ (\neg F_1 \wedge \neg F_2))$

Table 3: CTL equivalences.

ure 2(bottom), our example's termination requirement can more formally be represented as follows:

$$F \equiv \bigwedge_{i=1}^{n-1} \mathbf{term}(P_i) \leadsto \mathbf{term}(P_{i+1}).$$

Here we use the familiar $leads - to$ operator ("$\leadsto$"), which is defined in $\forall$CTL as

$$\mathcal{G}_1 \leadsto \mathcal{G}_2 \stackrel{\mathrm{def}}{=} \forall \Box (\mathcal{G}_1 \Longrightarrow \forall \Diamond (\mathcal{G}_2)).$$

One way or another, verifying this property entails: (1) considering the set of all states in which P_i terminates, and (2) determining whether every reachable path leaving those states contains a state where task P_{i+1} terminates.

The way we go about doing this is by converting the $\forall$CTL properties into duals, and then checking for counter-examples. We present the rules for such a conversion in Table 3. As can be seen from the table, the dual of a $\forall$CTL

Satisfaction of an $\exists$CTL sentence, F, by state, $\vec{x}$, wrt. Σ, the set of paths in the model.	
$F \equiv \exists \bigcirc G$	$\vec{x} \models F$ iff $\exists \sigma \in \Sigma$: $\vec{x} = \sigma(0) \wedge \sigma(1) \models G$
$F \equiv \exists \Diamond G$	$\vec{x} \models F$ iff $\exists \sigma \in \Sigma$: $\vec{x} = \sigma(0) \wedge (\exists j \geq 0 : \sigma(j) \models G)$
$F \equiv \exists \Box G$	$\vec{x} \models F$ iff $\exists \sigma \in \Sigma$: $\vec{x} = \sigma(0) \wedge (\forall j \geq 0 : \sigma(j) \models G)$
$F \equiv G\ \exists \mathcal{U}\ H$	$\vec{x} \models F$ iff $\exists \sigma \in \Sigma$: $\vec{x} = \sigma(0) \wedge (\exists j \geq 0 : \sigma(j) \models H \wedge$ $\forall k : 0 \leq k < j : \sigma(k) \models G)$

Table 4: $\exists$CTL logic semantics.

formula will always end up in $\exists$CTL, i.e., the existentially quantified sublanguage of CTL. The semantics of the $\exists$CTL temporal operators is given in Table 4.

Consider one of the conjuncts of our requirement F:

$$F_i \equiv \mathbf{term}(P_i) \leadsto \mathbf{term}(P_{i+1})$$

which possesses as its dual the $\exists$CTL sentence:

$$\neg F_i \equiv \exists \Diamond (\mathbf{term}(P_i) \wedge \exists \Box \neg \mathbf{term}(P_{i+1})).$$

Let $\vec{x_I}$ be the initial state of the model of our program, and let $\sigma(i)$ the ith state in the path σ. Then in the positive (universally quantified) logic, we are asking the question:

$$\forall \sigma : \sigma(0) = \vec{x_I} : \sigma \models F_i \quad ?$$

But instead, we will ask whether there are any counter-examples:

$$\exists \sigma : \sigma(0) = \vec{x_I} : \sigma \models \neg F_i \quad ?$$

This is often (though not always) a much less expensive question to answer.

3.2 Models

Consider a system of n tasks, in which the states of task i are drawn from a finite set

$$S_i = \{s_{i_1}, s_{i_2}, \ldots, s_{i_m}\}$$

where each s_{i_j} represents a potential control point in task i. On the other hand, a state in the full composition of *all* of the tasks will range over

$$S = S_1 \times S_2 \times \ldots \times S_n.$$

We denote elements of S by vectors such as

$$\vec{x_j} = (s_{1_{j1}}, s_{2_{j2}}, \ldots, s_{n_{jn}})$$

where each $s_{i_{j_i}} \in S_i$.

The set of atomic propositions, AP, consists of state labels introduced automatically during the model generation process as well as user defined labels that denote specific states in the execution of a task. We define a set of valuation functions for AP of the sort:

$$f : (S_1 \times S_2 \times \ldots \times S_n) \rightarrow \{\mathbf{True}, \mathbf{False}\}.$$

For each $f \in AP$ its complement $\neg f$ is also defined as an atomic proposition in AP such that $\neg f(\vec{x})$ maps to **True** (**False**) if and only if $f(\vec{x})$ maps to **False** (**True**).

3.3 Transitions

The global, closed system's transitions T range over $S \times S$, where if $(\vec{x}, \vec{x}') \in T$, the interpretation is that the system can move from $\vec{x}$ to $\vec{x}'$. We alternatively use the notation $\vec{x} \rightarrow \vec{x}'$ to denote the fact that $(\vec{x}, \vec{x}') \in T$.

In practice, of course, we usually do not deal with global transitions – after all, we are aiming for a compositional strategy, where we can perform our analysis incrementally.

When task i is compiled, its local transition model is formed, which we denote $\widehat{T}_i \subseteq S_i \times S_i$. This model is generated independently of the others, and an element (s_i, s_i') of $\widehat{T}_i$ does not include any information about the states of the other tasks.

However we can (and do) give $\widehat{T}_i$ meaning in the closed system as well. The local transition system can trivially be extended into a set of potential global transitions as follows:

$$T_i = \{(s_1, \ldots, s_i, \ldots, s_n) \rightarrow (s_1', \ldots, s_i', \ldots, s_n') \\ \mid (s_i, s_i') \in \widehat{T}_i \wedge \forall j : s_j, s_j' \in S_j\} .$$

Obviously we do not actually expand out this set T_i. Rather, when the analyzer generates a local state, it uses the "wild card" symbol ("$-$") to refer to the other parts of the system. For example, let $(s_i, s_i') \in \widehat{T}_i$, and consider the following symbolic transition:

$$(-, \ldots, -, s_i, -, \ldots, -) \rightarrow (-, \ldots, -, s_i', -, \ldots, -).$$

We use this to denote the set of all potential *global* transitions consistent with (s_i, s_i'), i.e.,

$$\{(s_1, \ldots, s_i, \ldots, s_n) \rightarrow (s_1', \ldots, s_i', \ldots, s_n') \mid \forall j : s_j, s_j' \in S_j\}.$$

As we go about composing transition systems, and performing model-checking on them, we refine these wild cards into real system states.

3.4 Composition: Theory and Practice

If two transition models, T_i and T_j, were completely independent, then getting their parallel composition would simply be a matter of taking their intersection, $T_i \cap T_j$. But in general, concurrent tasks are not independent – they share synchronization and communication constraints. We capture this by a set of $\forall$CTL formulae, the conjunction of which we call a *refinement relation*. A decision procedure "chops off" transitions inconsistent with the refinement relation, i.e., those not allowed by the language's semantics. The refinement relation itself is generated by the compiler; it uses the source program and a set of target-specific semantic rules to form each clause.

We demonstrate this concept on our example program. In Figure 3 we give a more detailed transition graph for task P_1. The labels on the states are atomic propositions, which are generated by the compiler. The interpretations for some of these atoms are as follows:

start$(task)$**:** true in $task$'s initial state.

begin_call$(task, entry)$**:** true in the states where $task$ is ready to begin a call on the specified $entry$.

active_call$(task, entry)$**:** true in the states where $task$'s call to the specified $entry$ has been accepted.

end_call$(task, entry)$**:** true in the states where $task$'s call to the specified $entry$ has just completed.

term$(task)$**:** true in the state associated with $task$'s termination.

Consider the "Ready" rendezvous between tasks P_1 and P_2. If P_1 and P_2 are both prepared to rendezvous, then they will be in states labeled **begin_call**$(P_1, P_2.\text{Ready})$ (P_1's state 5), and **begin_accept**(P_2, Ready) (P_2's state 5), respectively. The rendezvous occurs when P_1 and P_2 take the transitions to the states labeled **active_call**$(P_1, P_2.\text{Ready})$ (P_1's state 6), and **active_accept**(P_2, Ready) (P_2's state 6), respectively. Ada's semantics insist that the rendezvous will occur if and only if both tasks commit to it, i.e., if and only if both tasks take their local transitions $5 \rightarrow 6$.

But the intersection of these two models includes transitions that are counter to the semantics, e.g.:

$$(5, 5, \ldots) \rightarrow (6, 5, \ldots) \in T_1 \cap T_2 .$$

That is, P_1 autonomously decides to rendezvous whereas P_2 decides to wait. Such a transition is deleted in our refinement check.

The refinement relation is built by doing a simple scan of all the tasks in the program, and extracting their inter-task interfaces. Then, from the interface data, the analyzer determines which tasks synchronize with which other tasks, and which propositions must hold when such synchronizations occur.

The entire relation, $\mathcal{R}$ is specified via a CTL formula. The CTL formula specifying $\mathcal{R}$ is the conjunction of subformulae in the form $f \rightarrow g$ or $f \rightarrow \forall \bigcirc (g)$, where f and g are propositions containing only the atomic propositions

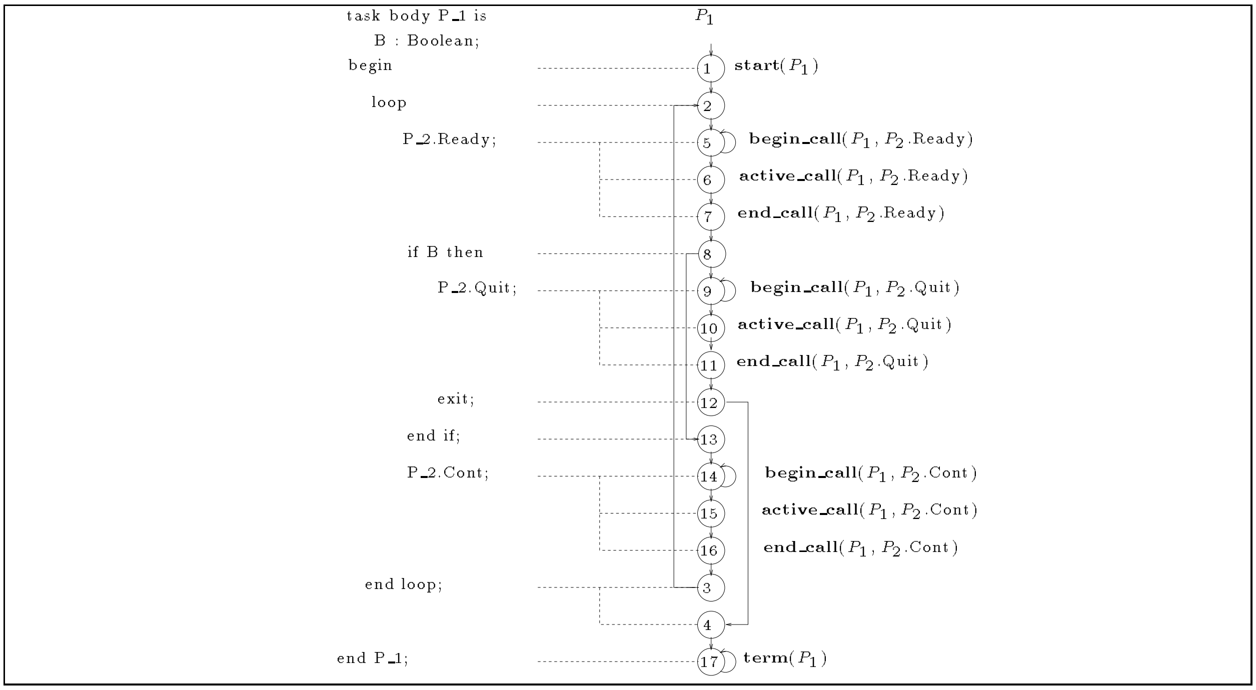

Figure 3: Source code and transition relation graph of the first task in the n task chain example.

and boolean connectives. In Figure 4, we show the part of $\mathcal{R}$ which refines the transition systems with respect to task P_i. The complete refinement relation will be the conjunction of similar formulae for each task. Explanations for two of the subformulae (annotated in Figure 4) are given below:

1 If P_{i-1} is ready to call "P_i.Ready," and if P_i is ready to "accept Ready," then in all next states both P_{i-1} and P_i are in the rendezvous.

2 If task P_i is in rendezvous "P_i+1.Ready," then task P_{i+1} must be in the rendezvous at its "accept Ready."

The parallel composition of T_i and T_j can be defined as follows:

$$T_i \| T_j \stackrel{\text{def}}{=} T_i \cap T_j \cap T_{\mathcal{R}}$$

where $T_{\mathcal{R}}$ is the largest set of transitions satisfying $\mathcal{R}$. The algorithm in Figure 5 effectively computes this composed transition model.

3.5 Model Checking

Our compositional model-checker does not try to make exact decisions about a property (which usually requires a complete model). Rather, it incrementally refines estimates using partial models. These estimates are conservative, in that if any counter-examples to the specified property exist then the partial models we return from our checks will always contain them. As a result, if any check on a partial model fails to produce a potential counter-example, then we can immediately infer that the complete model does not have any either. To decide the subset of a model that satisfies a formula we recursively apply the *projection functions* shown in Table 5. We note that while these functions yield estimates for partial compositions, e.g., $T_i \parallel T_j$, they give exact results when applied to a composition that includes all components, e.g., T.

Consider evaluating an atomic proposition on the transition relation T_p. Assume that T_p is a partial model, i.e., $p \subset \{1, \ldots, n\}$. Then a state $\vec{x_p}$ of this model is of the form:

$$\vec{x_p} = (s_{p_1}, \ldots, s_{p_m}, -, \ldots, -)$$

where m is the cardinality of p. This model does not give us any information about the states of the tasks which are not in this partial model. The only information we have about the non-local models (i.e., the models which are not in set p) is the synchronization constraints defined by $\mathcal{R}$. So, it may be the case that when we try to evaluate an atomic proposition f on $\vec{x_p}$, both $f(\vec{x_p})$ and $\neg f(\vec{x_p})$ are **True**. Consequently, the projection function could decide a transition, with $\vec{x_p}$ as its source state, potentially satisfies both f and $\neg f$. Now

	Refinement Formula for Task P_i (Imposes Ada Semantics)
	$(\mathbf{begin\_call}(P_{i-1}, P_i.\text{Ready}) \wedge \mathbf{begin\_accept}(P_i, \text{Ready})) \rightarrow \forall \bigcirc (\mathbf{active\_call}(P_{i-1}, \text{Ready}) \wedge \mathbf{active\_accept}(P_i, \text{Ready}))$ [1]
$\wedge$	$(\mathbf{begin\_accept}(P_i, \text{Ready}) \wedge \neg \mathbf{begin\_call}(P_{i-1}, P_i.\text{Ready})) \rightarrow \forall \bigcirc (\mathbf{begin\_accept}(P_i, \text{Ready}))$
$\wedge$	$(\mathbf{active\_call}(P_i, P_{i+1}.\text{Ready}) \wedge \mathbf{active\_accept}(P_{i+1}, \text{Ready})) \rightarrow \forall \bigcirc (\mathbf{end\_call}(P_i, P_{i+1}.\text{Ready}))$
$\wedge$	$(\mathbf{begin\_call}(P_i, P_{i+1}.\text{Ready}) \wedge \neg \mathbf{begin\_accept}(P_{i+1}, \text{Ready})) \rightarrow \forall \bigcirc (\mathbf{begin\_call}(P_i, P_{i+1}.\text{Ready}))$
$\wedge$	$(\mathbf{begin\_call}(P_{i-1}, P_i.\text{Quit}) \wedge \mathbf{begin\_accept}(P_i, \text{Quit})) \rightarrow \forall \bigcirc (\mathbf{active\_call}(P_{i-1}, \text{Quit}) \wedge \mathbf{active\_accept}(P_i, \text{Quit}))$
$\wedge$	$(\mathbf{begin\_accept}(P_i, \text{Quit}) \wedge \neg \mathbf{begin\_call}(P_{i-1}, P_i.\text{Quit})) \rightarrow \forall \bigcirc (\mathbf{begin\_accept}(P_i, \text{Quit}))$
$\wedge$	$(\mathbf{active\_call}(P_i, P_{i+1}.\text{Quit}) \wedge \mathbf{active\_accept}(P_{i+1}, \text{Quit})) \rightarrow \forall \bigcirc (\mathbf{end\_call}(P_i, P_{i+1}.\text{Quit}))$
$\wedge$	$(\mathbf{begin\_call}(P_i, P_{i+1}.\text{Quit}) \wedge \neg \mathbf{begin\_accept}(P_{i+1}, \text{Quit})) \rightarrow \forall \bigcirc (\mathbf{begin\_call}(P_i, P_{i+1}.\text{Quit}))$
$\wedge$	$(\mathbf{begin\_call}(P_{i-1}, P_i.\text{Cont}) \wedge \mathbf{begin\_accept}(P_i, \text{Cont})) \rightarrow \forall \bigcirc (\mathbf{active\_call}(P_{i-1}, \text{Cont}) \wedge \mathbf{active\_accept}(P_i, \text{Cont}))$
$\wedge$	$(\mathbf{begin\_accept}(P_i, \text{Cont}) \wedge \neg \mathbf{begin\_call}(P_{i-1}, P_i.\text{Cont})) \rightarrow \forall \bigcirc (\mathbf{begin\_accept}(P_i, \text{Cont}))$
$\wedge$	$(\mathbf{active\_call}(P_i, P_{i+1}.\text{Cont}) \wedge \mathbf{active\_accept}(P_{i+1}, \text{Cont})) \rightarrow \forall \bigcirc (\mathbf{end\_call}(P_i, P_{i+1}.\text{Cont}))$
$\wedge$	$(\mathbf{begin\_call}(P_i, P_{i+1}.\text{Cont}) \wedge \neg \mathbf{begin\_accept}(P_{i+1}, \text{Cont})) \rightarrow \forall \bigcirc (\mathbf{begin\_call}(P_i, P_{i+1}.\text{Cont}))$
$\wedge$	$(\mathbf{at\_select}(P_i) \wedge (\mathbf{begin\_call}(P_{i-1}, P_i.\text{Quit}) \vee \mathbf{begin\_call}(P_{i-1}, P_i.\text{Cont}))) \rightarrow$
	$\forall \bigcirc ((\mathbf{begin\_accept}(P_i, \text{Quit}) \wedge \mathbf{begin\_call}(P_{i-1}, P_i.\text{Quit})) \vee (\mathbf{begin\_accept}(P_i, \text{Cont}) \wedge \mathbf{begin\_call}(P_{i-1}, P_i.\text{Cont})))$
$\wedge$	$(\mathbf{active\_accept}(P_i, \text{Ready}) \rightarrow \forall \bigcirc (\mathbf{end\_accept}(P_i, \text{Ready})) \quad \wedge \quad (\mathbf{active\_accept}(P_i, \text{Ready}) \rightarrow \mathbf{active\_call}(P_{i-1}, P_i.\text{Ready}))$
$\wedge$	$(\mathbf{end\_accept}(P_i, \text{Ready}) \rightarrow \mathbf{end\_call}(P_{i-1}, P_i.\text{Ready}))$
$\wedge$	$\mathbf{end\_call}(P_i, P_{i+1}.\text{Ready}) \rightarrow \mathbf{end\_accept}(P_{i+1}, \text{Ready}) \quad \wedge \quad \mathbf{active\_call}(P_i, P_{i+1}.\text{Ready}) \rightarrow \mathbf{active\_accept}(P_{i+1}, \text{Ready})$ [2]
$\wedge$	$\mathbf{active\_accept}(P_i, \text{Quit}) \rightarrow \forall \bigcirc (\mathbf{end\_accept}(P_i, \text{Quit})) \quad \wedge \quad \mathbf{active\_accept}(P_i, \text{Quit}) \rightarrow \mathbf{active\_call}(P_{i-1}, P_i.\text{Quit})$
$\wedge$	$\mathbf{end\_accept}(P_i, \text{Quit}) \rightarrow \mathbf{end\_call}(P_{i-1}, P_i.\text{Quit}) \quad \wedge \quad \mathbf{begin\_accept}(P_i, \text{Quit}) \rightarrow \mathbf{begin\_call}(P_{i-1}, \text{Quit})$
$\wedge$	$\mathbf{end\_call}(P_i, P_{i+1}.\text{Quit}) \rightarrow \mathbf{end\_accept}(P_{i+1}, \text{Quit}) \quad \wedge \quad \mathbf{active\_call}(P_i, P_{i+1}.\text{Quit}) \rightarrow \mathbf{active\_accept}(P_{i+1}, \text{Quit})$
$\wedge$	$\mathbf{active\_accept}(P_i, \text{Cont}) \rightarrow \forall \bigcirc (\mathbf{end\_accept}(P_i, \text{Cont})) \quad \wedge \quad \mathbf{active\_accept}(P_i, \text{Cont}) \rightarrow \mathbf{active\_call}(P_{i-1}, P_i.\text{Cont})$
$\wedge$	$\mathbf{end\_accept}(P_i, \text{Cont}) \rightarrow \mathbf{end\_call}(P_{i-1}, P_i.\text{Cont}) \quad \wedge \quad \mathbf{begin\_accept}(P_i, \text{Cont}) \rightarrow \mathbf{begin\_call}(P_{i-1}, \text{Cont})$
$\wedge$	$\mathbf{end\_call}(P_i, P_{i+1}.\text{Cont}) \rightarrow \mathbf{end\_accept}(P_{i+1}, \text{Cont}) \quad \wedge \quad \mathbf{active\_call}(P_i, P_{i+1}.\text{Cont}) \rightarrow \mathbf{active\_accept}(P_{i+1}, \text{Cont})$

Figure 4: Automatically generated refinement formula for task P_i.

function Compose_And_Refine(T_i,T_j : model, $\mathcal{R}$: refinement) : model
    T : model = $\emptyset$
begin
    foreach $(\vec{x}, \vec{x}') \in T_i \cap T_j$
        ok = true
        foreach $f \rightarrow g$ in $\mathcal{R}$
            – – Type(f) = Proposition (i.e., f contains only atoms and boolean connectives)
            if Satisfies($\vec{x}$,f) then
                case Type(g) of
                    $\forall \bigcirc (h)$:
                        – – Type(h) = Proposition
                        if not Satisfies($\vec{x}'$,h) then ok = false
                    Proposition:
                        if not Satisfies($\vec{x}$,g) then ok = false
                end case
            end if
        end foreach
        if ok then $T = T \cup \{(\vec{x}, \vec{x}')\}$
    end foreach
    return T
end Compose_And_Refine

Figure 5: Parallel composition.

$\exists$CTL Subformula $\mathcal{F}$	$projection(T, \mathcal{F})$
$f \in AP$	$\{(\vec{x_1}, \vec{x_2}) \in T \mid f(\vec{x_1}) \wedge (\vec{x_1}, \vec{x_2}) \in T_{\mathcal{R}}\}$
$f \wedge g$	$projection(T, f) \cap projection(T, g)$
$f \vee g$	$projection(T, f) \cup projection(T, g)$
$\exists \bigcirc f$	$\{(\vec{x_1}, \vec{x_2}) \in T \mid \exists(\vec{x_1}, \vec{x_3}) \in T$ $\wedge \exists(\vec{x_3}, \vec{x_4}) \in projection(T, f)\}$
$\exists \Box f$	$\{(\vec{x_1}, \vec{x_2}) \in T \mid \exists(\vec{x_1}, \vec{x_3}) \in fix_n T_n\}$ with $fix_n T_n : T_0 = projection(T, f)$ $T_{n+1} = \{(\vec{x_1}, \vec{x_2}) \in T_n$ $\mid \exists(\vec{x_2}, \vec{x_3}) \in T_n\}$
$f \exists \mathcal{U} g$	$\{(\vec{x_1}, \vec{x_2}) \in T \mid \exists(\vec{x_1}, \vec{x_3}) \in fix_n T_n\}$ with $fix_n T_n : T_0 = projection(T, g)$ $T_{n+1} = \{(\vec{x_1}, \vec{x_2}) \in projection(T, f)$ $\mid \exists(\vec{x_2}, \vec{x_3}) \in T_n\} \ \cup \ T_n$

Table 5: Projection functions for $\exists$CTL.

assume that T_p is not a partial model. Then it is the case that the state is completely specified:

$$\vec{x_p} = (s_1, \ldots, s_i, \ldots, s_n)$$

and so only one of f and $\neg f$ can be true, and so in this case the projection function returns the exact result.

Another crucial point in the evaluation of atomic propositions on partial models is the usage of the refinement relation $\mathcal{R}$. As can be seen in the first row of Table 5, projection function of an atomic proposition f includes transitions which satisfy two conditions: 1) f evaluates to **True** for the source state and 2) the transition is consistent with the refinement relation. If we have a partial model T_p as defined above, then these conditions may be satisfied by the transition $(\vec{x_p}, \vec{x_p}')$ by substituting non-local states for the don't care conditions "$-$". For example,

$$f(\vec{x_p}) = \textbf{True} \ \textbf{ iff } \ \exists \ s_{j_1}, \ldots, s_{j_{n-m}} \ : \ j_1, \ldots, j_{n-m} \notin p \ : \ f(s_{p_1}, \ldots, s_{p_m}, s_{j_1}, \ldots, s_{j_{n-m}}) = \textbf{True}.$$

The second condition requires that the same substitutions for the non-local states should be consistent with the refinement relation. Hence, the the complete condition is :

$$\begin{gathered} \exists \ s_{j_1}, \ldots, s_{j_{n-m}} \ : \ j_1, \ldots, j_{n-m} \notin p \ : \\ f(s_{p_1}, \ldots, s_{p_m}, s_{j_1}, \ldots, s_{j_{n-m}}) = \textbf{True} \ \wedge \\ ((s_{p_1}, \ldots, s_{p_m}, s_{j_1}, \ldots, s_{j_{n-m}}), x_p') \in T_{\mathcal{R}}. \end{gathered}$$

This mechanism enables us to remove the transitions which are inconsistent with atomic proposition f according to the refinement relation. We implement this as follows: when we are computing projection of an atomic proposition we pull out the subformulae of the refinement relation which involves that particular atomic proposition. We check if the local states of the transition are consistent with these subformulae.

The evaluation of the other projection functions is straight-forward. Consider the evaluation of a $\exists\Box(f)$ formula wrt. the transition relation T. We first identify $T_0 \subseteq T$ where f holds in the source state. Then, from a set of transitions T_{n-1}, we construct a set T_n by removing transitions with destination states that are not in T_{n-1}. Obviously the fixpoint is reached when deletion stops; i.e., when T_{n+1} is found equal to T_n. The idea is similar for $f \ \exists\mathcal{U} \ g$. We discover all of those transitions where g is true in the source state. These transitions automatically satisfy $f \ \exists\mathcal{U} \ g$. Then we work our way backwards, constructing paths to this base set over which f is always true.

The complete compositional model checking approach is summarized in Figure 6.

4 Compositional Model Checking Toolset

The toolset is best illustrated by applying it to an example, in this case the n task chain example given in section 2. Consider the property mentioned in section 2:

> *Whenever task $P_{i,i<n}$ has terminated, task P_{i+1} will eventually terminate.*

Or in CTL,

$$F \equiv \bigwedge_{i=1}^{n-1} \mathbf{term}(P_i) \leadsto \mathbf{term}(P_{i+1}).$$

To show that $(P_1 \| P_2 \| \ldots \| P_n) \models F$, we divide this problem so that it can be attacked compositionally. Let F_i be defined as follows:

$$F_i \equiv \mathbf{term}(P_i) \leadsto \mathbf{term}(P_{i+1})$$

where $1 \leq i < n$. Hence, $F \equiv \wedge_{i=1}^{n-1} F_i$.

We use the following three properties of CTL to make deductions from the formulae checked by the model checker:

$$\frac{\mathcal{F} \leadsto \mathcal{G} \quad \mathcal{G} \leadsto \mathcal{H}}{\mathcal{F} \leadsto \mathcal{H}} \ (1) \qquad \frac{\forall\Box(\mathcal{F} \Longrightarrow \forall\Box\mathcal{G}) \quad \mathcal{F} \leadsto \mathcal{H}}{\mathcal{F} \leadsto (\mathcal{G} \wedge \mathcal{H})} \ (2) \qquad \frac{\textbf{True} \leadsto \mathcal{G}}{\mathcal{F} \leadsto \mathcal{G}}. \ (3)$$

All three properties can be derived from the base rules for CTL. We can use property (1) to strengthen the antecedent

1. FOR each Ada tasking program DO
   (a) Translate the Ada task into a finite transition system.
   (b) Remove transitions that are semantically inconsistent or unreachable.
2. SELECT a specification stated as a ∀CTL sentence.
   (a) Parse the selected sentence, negate it, and emit a postfix form ∃CTL sentence.
3. WHILE there are any unchecked or uncomposed models DO
   (a) Either CHOOSE an unchecked model, removing it from the set of models and
      - Decide the subset of the selected model that (potentially) satisfies the sentence.
      - Remove non-satisfying transitions, creating a new model.
   (b) or CHOOSE some models, removing them from the set of models and
      - Compose the models and refine the composition creating a new model.
   (c) IF the new model is empty THEN RETURN **True** ELSE add the new model to the set of models.
4. RETURN **False**. (The remaining model contains the counter-examples.)

Figure 6: Compositional model checking procedure.

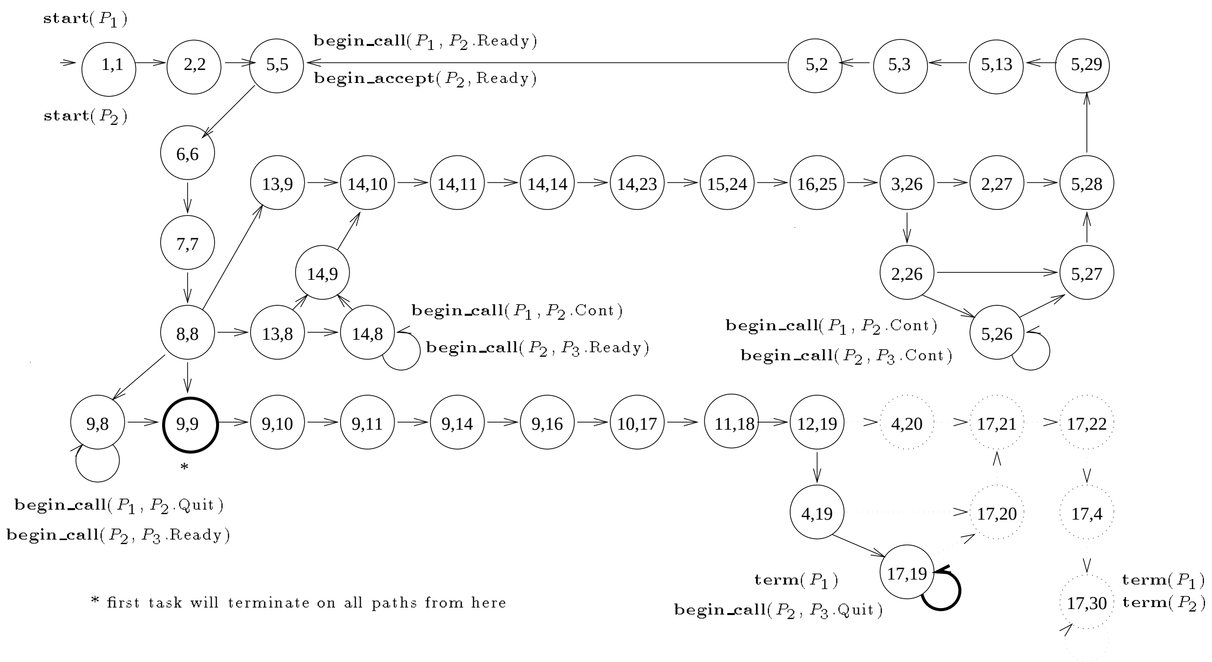

Figure 7: Composition of P_1 and P_2.

in the implication if we are not able to prove the formula $\mathcal{F} \leadsto \mathcal{H}$ with the model checker. Assume that, we substitute $\mathcal{F} \wedge \mathcal{A}$ for $\mathcal{G}$ in property (1), where $\mathcal{A}$ is the strengthening assertion. If strengthening assertion $\mathcal{A}$ removes the transitions which create potential counter-examples, we will be able to prove the property $(\mathcal{F} \wedge \mathcal{A}) \leadsto \mathcal{H}$ using the model checker. If we can also prove $\mathcal{F} \leadsto (\mathcal{F} \wedge \mathcal{A})$, then using property (1) we can deduce $\mathcal{F} \leadsto \mathcal{H}$.

Steps of the analysis are summarized in Figure 8, where justification of each step is written in the right column. The lines which are marked "*" indicate proof obligations, i.e. formulae which are proved in following steps.

Step 1.1 : We note that the sentence F_1 only constrains tasks P_1 and P_2; many (possibly all) states of the other $n-2$ tasks might be consistent with both F_1 and $\neg F_1$. This fact naturally yields the following strategy: to first run $\mathbf{check}(P_1 \| P_2, \neg F_1)$, and see if the result is $\emptyset$. If the returned set of states is non-empty, then it contains the potential counter-example to F_1.

In our case the result is nonempty. Figure 7 shows the transition system corresponding to $P_1 \| P_2$ which has 43 states and 55 transitions. When we check $\neg F_1$ on $P_1 \| P_2$ we get a minor reduction. The submodel returned by $\mathbf{check}(P_1 \| P_2, \neg F_1)$ is drawn with solid lines in Figure 7, and it has 37 states and 46 transitions. These may ultimately be shown to be unreachable behaviors, restricted by interactions with the other tasks. At this point they must be retained as "candidate" counter-examples, and they serve to guide the next steps in our analysis.

Consider the composite state (9,9) (marked with * in Figure 7). The state corresponds to P_1 having signalled its readiness to terminate, after which P_2 will prepare to terminate as well. But this cannot happen until P_3 allows it, and so on. In fact, the self-loop on composite state (17,19) is the *critical* transition for a counter-example, in that $\neg F_1$ holds if and only if the self-loop is taken infinitely often. The loop corresponds to P_3's refusal to accept P_2's call; if we can prove that P_3 accepts the call, then we will have no counter-example to F_1.

Now we can show how strengthening assertions can aid in the compositional approach. Consider property F_1 : "*Whenever P_1 has terminated, P_2 will eventually terminate.*" We will strengthen it with F_1' : "*Whenever P_1 has terminated and P_2 is not waiting for its calls to be accepted, P_2 will eventually terminate.*" F_1' can be stated as:

$$F_1' \equiv (\mathbf{term}(P_1) \wedge A_2) \leadsto \mathbf{term}(P_2)$$

where

$$\begin{aligned} A_2 \quad \equiv \quad & \neg\mathbf{wait\_call}(P_2, P_3.\mathrm{Ready}) \wedge \\ & \neg\mathbf{wait\_call}(P_2, P_3.\mathrm{Quit}) \wedge \\ & \neg\mathbf{wait\_call}(P_2, P_3.\mathrm{Cont}). \end{aligned}$$

Here, $\mathbf{wait\_call}(P_i, P_{i+1}.X)$ $(X \in \{\mathrm{Ready}, \mathrm{Quit}, \mathrm{Cont}\})$ is defined as

$$\begin{aligned} \mathbf{wait\_call}(P_i, P_{i+1}.X) \quad \equiv \quad & \mathbf{begin\_call}(P_i, P_{i+1}.X) \wedge \\ & \neg\mathbf{begin\_accept}(P_{i+1}, X). \end{aligned}$$

So the deductive step which follows from property (1) is:

$$\frac{\begin{array}{l}\mathbf{term}(P_1) \leadsto (\mathbf{term}(P_1) \wedge A_2) \\ F_1' \equiv (\mathbf{term}(P_1) \wedge A_2) \leadsto \mathbf{term}(P_2)\end{array}}{F_1 \equiv \mathbf{term}(P_1) \leadsto \mathbf{term}(P_2).}$$

We call the model checker to verify F_1' on the composition $P_1 \| P_2$, and as can be seen in Figure 9's dialog box, $\mathbf{check}(P_1 \| P_2, \neg F_1') = \emptyset$; so F_1' holds.

Step 1.2 : Now we have to prove $\mathbf{term}(P_1) \leadsto (\mathbf{term}(P_1) \wedge A_2)$. To prove it we will use property (2) as follows:

$$\frac{\begin{array}{l}\forall\Box(\mathbf{term}(P_1) \Longrightarrow \forall\Box(\mathbf{term}(P_1))) \\ \mathbf{term}(P_1) \leadsto A_2\end{array}}{\mathbf{term}(P_1) \leadsto (\mathbf{term}(P_1) \wedge A_2).}$$

We check $\forall\Box(\mathbf{term}(P_1) \Longrightarrow \forall\Box(\mathbf{term}(P_1)))$, and do not get any counter-examples, hence it holds. I.e., once P_1 terminates, it remains terminated.

Step 1.3 : Instead of proving $\mathbf{term}(P_1) \leadsto A_2$, we prove a stronger property: "*All of P_2's calls are eventually accepted.*" That is, A_2 eventually holds. Formally, we want to prove the property $I_2 \equiv \mathbf{True} \leadsto A_2 \equiv \forall\Box\forall\Diamond A_2$. If we can prove this property, then $\mathbf{term}(P_1) \leadsto A_2$ is also true, since the consequent of the implication is always true (property (3)). Then using property (2) we can deduce $\mathbf{term}(P_1) \leadsto (\mathbf{term}(P_1) \wedge A_2)$, as shown above. This will also allow us to deduce $F_1 \equiv \mathbf{term}(P_1) \leadsto \mathbf{term}(P_2)$ using property (1) as explained above.

If we try to prove the property I_2 on the composition $P_1 \| P_2$ we will not be successful because I_2 is related to the behavior of P_3. So, as the next step we compose P_2 and P_3 on which we will try to prove F_2 and I_2.

Steps 2.1 - 2.3 : These steps are identical to steps 1.1 - 1.3. We first check F_2' on $P_2 \| P_3$ and get an empty transition system as the result, so F_2' holds. If we can show that I_3

$F_i \equiv \mathbf{term}(P_i) \leadsto \mathbf{term}(P_{i+1})$
$A_i \equiv \neg\mathbf{wait\_call}(P_i, P_{i+1}.\text{Ready}) \wedge \neg\mathbf{wait\_call}(P_i, P_{i+1}.\text{Quit}) \wedge \neg\mathbf{wait\_call}(P_i, P_{i+1}.\text{Cont})$
$\mathbf{wait\_call}(P_i, P_{i+1}.X) \equiv \mathbf{begin\_call}(P_i, P_{i+1}.X) \wedge \neg\mathbf{begin\_accept}(P_{i+1}, X)$ where $X \in \{\text{Ready}, \text{Quit}, \text{Cont}\}$
$F_i' \equiv (\mathbf{term}(P_i) \wedge A_{i+1}) \leadsto \mathbf{term}(P_{i+1})$
$I_i \equiv \mathbf{True} \leadsto A_i \equiv \forall\Box\forall\Diamond A_i$
$I_i' \equiv A_{i+1} \leadsto A_i$
$T_i \equiv \forall\Box(\mathbf{term}(P_i) \Longrightarrow \forall\Box(\mathbf{term}(P_i)))$

	FORMULA	JUSTIFICATION
1.1	$\mathbf{term}(P_1) \leadsto (\mathbf{term}(P_1) \wedge A_2)$	* (Proved in step **1.2**)
	$F_1' \equiv (\mathbf{term}(P_1) \wedge A_2) \leadsto \mathbf{term}(P_2)$	$\boxed{\mathbf{check}(P_1 \| P_2, \neg F_1') = \emptyset}$
	$F_1 \equiv \mathbf{term}(P_1) \leadsto \mathbf{term}(P_2)$	Property (1)
1.2	$T_1 \equiv \forall\Box(\mathbf{term}(P_1) \Longrightarrow \forall\Box(\mathbf{term}(P_1)))$	$\boxed{\mathbf{check}(P_1, \neg T_1) = \emptyset}$
	$\mathbf{term}(P_1) \leadsto A_2$	* (Proved in step **1.3**)
	$\mathbf{term}(P_1) \leadsto (\mathbf{term}(P_1) \wedge A_2)$	Property (2)
1.3	$I_2 \equiv \mathbf{True} \leadsto A_2$	* (Proved in step **2.4**)
	$Term(P_1) \leadsto A_2$	Property (3)
	Remaining proof obligations from step 1 : $I_2 \equiv \mathbf{True} \leadsto A_2$	
2.1	$\mathbf{term}(P_2) \leadsto (\mathbf{term}(P_2) \wedge A_3)$	* (Proved in step **2.2**)
	$F_2' \equiv (\mathbf{term}(P_2) \wedge A_3) \leadsto \mathbf{term}(P_3)$	$\boxed{\mathbf{check}(P_2 \| P_3, \neg F_2') = \emptyset}$
	$F_2 \equiv \mathbf{term}(P_2) \leadsto \mathbf{term}(P_3)$	Property (1)
2.2	$T_2 \equiv \forall\Box(\mathbf{term}(P_2) \Longrightarrow \forall\Box(\mathbf{term}(P_2)))$	$\boxed{\mathbf{check}(P_2, \neg T_2) = \emptyset}$
	$\mathbf{term}(P_2) \leadsto A_3$	* (Proved in step **2.3**)
	$\mathbf{term}(P_2) \leadsto (\mathbf{term}(P_2) \wedge A_3)$	Property (2)
2.3	$I_3 \equiv \mathbf{True} \leadsto A_3$	* (Proved in step **3.4**)
	$\mathbf{term}(P_2) \leadsto A_3$	Property (3)
2.4	$I_3 \equiv \mathbf{True} \leadsto A_3$	* (Proved in step **3.4**)
	$I_2' \equiv A_3 \leadsto A_2$	$\boxed{\mathbf{check}(P_2 \| P_3, \neg I_2') = \emptyset}$
	$I_2 \equiv \mathbf{True} \leadsto A_2$	Property (1)
	Remaining proof obligations from step 2 : $I_3 \equiv \mathbf{True} \leadsto A_3$	
.		
.		
.		
	Remaining proof obligations from step $(n-2)$: $I_{n-1} \equiv \mathbf{True} \leadsto A_{n-1}$	
$(n-1)$.1	$F_{n-1} \equiv \mathbf{term}(P_{n-1}) \leadsto \mathbf{term}(P_n)$	$\boxed{\mathbf{check}(P_{n-1} \| P_n, \neg F_{n-1}) = \emptyset}$
$(n-1)$.2	$I_{n-1} \equiv \mathbf{True} \leadsto A_{n-1}$	$\boxed{\mathbf{check}(P_{n-1} \| P_n, \neg I_{n-1}) = \emptyset}$

Figure 8: Summary of the analysis.

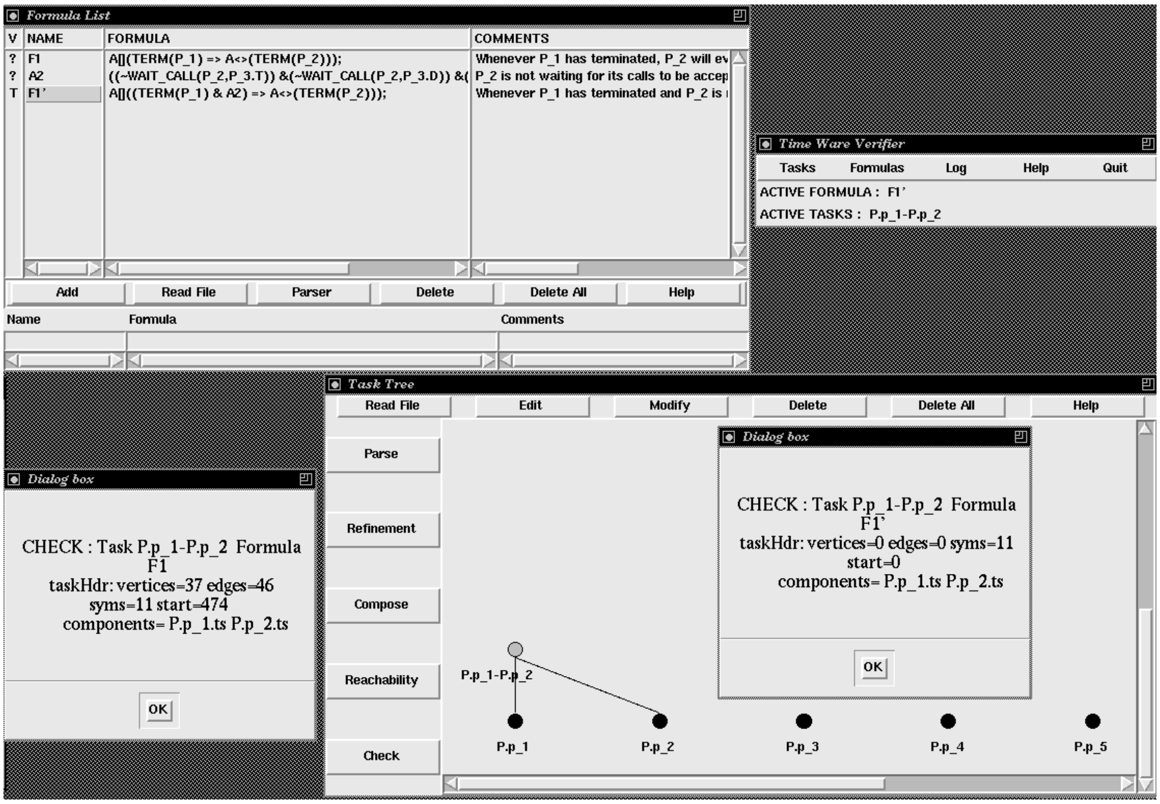

Figure 9: Results of **check**$(P_1 \| P_2, \neg F_1)$ and **check**$(P_1 \| P_2, \neg F_1')$.

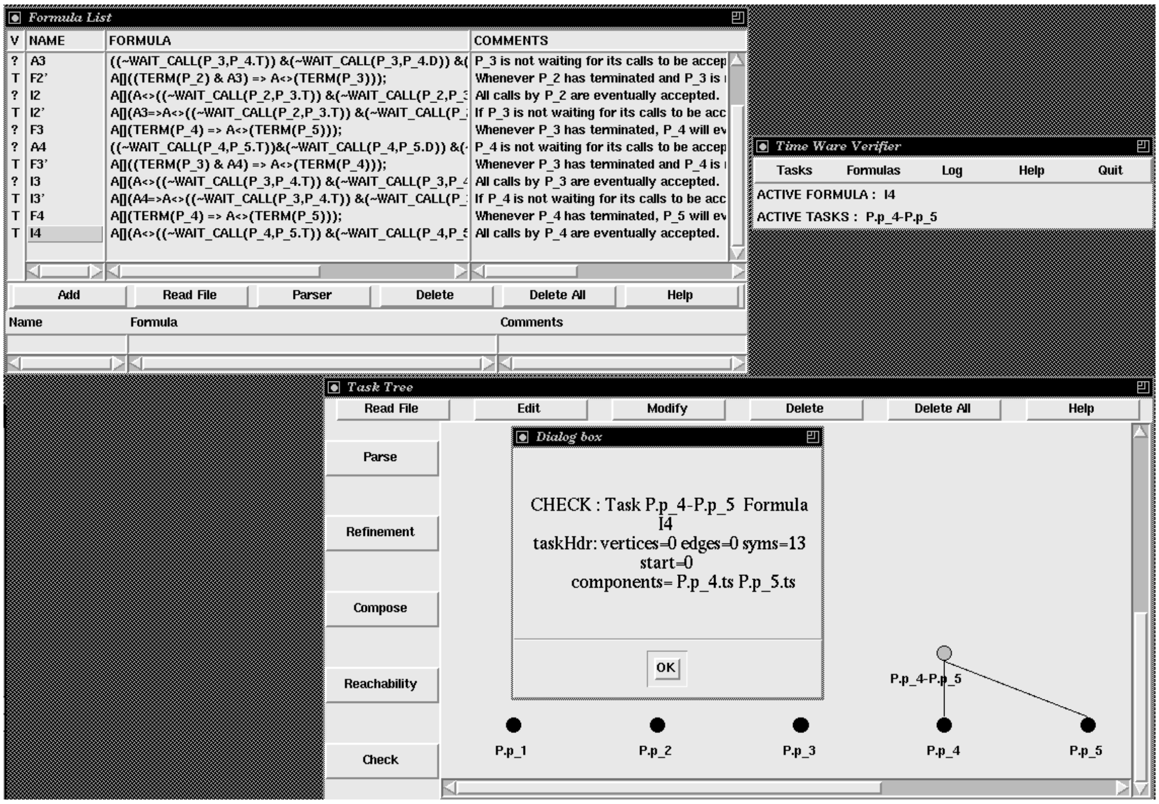

Figure 10: Result of **check**$(P_4 \| P_5, \neg I_4)$.

holds, then from F_2' and I_3 we can deduce F_2, just as we did for F_1. So I_3 is the proof obligation in these steps.

Step 2.4 : Now we have to prove the proof obligation from the previous step, i.e. I_2. If we try to prove the property I_2 on $P_2 \| P_3$ we will have the same problem we had for F_1. So we use the strengthening argument again. Instead of proving I_2 we try to prove $I_2' \equiv A_3 \leadsto A_2$. Then using property (1):

$$\frac{\begin{array}{l} I_3 \equiv \mathbf{True} \leadsto A_3 \\ I_2' \equiv A_3 \leadsto A_2 \end{array}}{I_2 \equiv \mathbf{True} \leadsto A_2}$$

we deduce I_2. When we call $\mathbf{check}(P_2 \| P_3, \neg I_2')$, we get $\emptyset$ as the result, so I_2' holds.

Steps 3 - $(n-2)$: If we continue to repeat the operations discussed above we will get the same results until the $n-1$st step, since the intermediate $n-2$ tasks are identical.

Step $(n-1)$: In the $(n-1)$st step we notice that we do not need the strengthening assertions anymore! The reason is that we reached the end of the task chain, and P_n does not have to wait for any other task. It will accept all the calls from P_{n-1}. So we first call $\mathbf{check}(P_{n-1} \| P_n, \neg F_{n-1})$ and get $\emptyset$ as the result, so F_{n-1} holds. Now we have to prove the last proof obligation from the previous step, i.e. I_{n-1}, which will complete the proof. Figure 10 shows the result of $\mathbf{check}(P_{n-1} \| P_n, \neg I_{n-1})$ for $n = 5$, and it is $\emptyset$, so I_{n-1} is satisfied, hence the proof is complete.

Note that, the proof given above is a semi-automated process. We designed the structure of the proof by examining the counter-example output by the model checker. After coming up with the strengthening assertions (which still needs human interaction) we used the model checker to verify the formulae in each step. We did not have to deal with the details involved in verifying these formulae, which may be very tedious.

5 Results and Future Work

While the introduction of our compositional approach has not increased the worst case complexity, if the worst case is realized then we haven't gained anything over the traditional approach. We believe, however, that for many systems and properties our approach can do dramatically better than the traditional approach.

Figure 11 illustrates the tremendous savings that can sometimes be provided by our compositional approach. Here we are using the total number of transitions that must be examined and/or composed as the cost metric. In the case of the n task chain, when following the traditional approach our composition tool ran out of memory even for small values of n. But with our compositional method we never need to create such large n-way compositions to perform the verification, for *any* n.

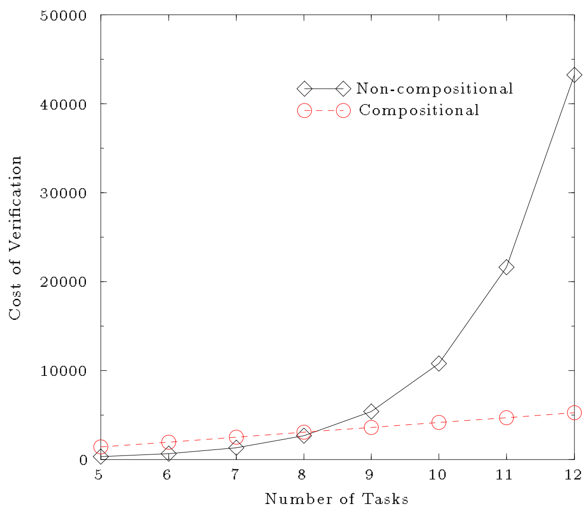

Figure 11: Cost of compositional and non-compositional model-checks of n task chain.

Our experiments have shown that there can be a large variance in the total cost of checking. The variance is caused by differences in models, formulas, and order of operations. E.g., given two compositional checks of the same sentence, and on the same components, the total cost can be orders of magnitude apart! As part of our on-going work we are seeking to develop heuristics for generating compositional model checking schedules that, given a sentence and a model, would suggest:

- Which sentence(s) to check : Should we break the original sentence down into a set of conjuncts or disjuncts, and then check each new sentence individually?

- The order of checks and compositions : Which models should be checked first? Which models should be checked last? Which models should we compose before checking?

- Adjustments to the schedule based upon feedback : What strengthening assertions are suggested by the potential counter-examples identified by the model checker? If checking a model yields a very large reduction, then should we immediately compose the result with another model: If so, which model? If we see our compositions are blowing up, should we backup and take a different scheduling path?

Even if scheduling guidance is not incorporated into the tool itself, we see the development of some *rules of thumb* as being critical to the practical application of a compositional approach.

As part of our future work we intend to add real-time modelling capabilities. While there has been recent research in real-time model checking (see [1, 12]), there has been very little exploration using a compositional framework. The addition of time makes this a significantly more complex problem.

6 Conclusions

We have developed an automated approach to verification that is compositional. As we have demonstrated with an example, our approach enables us to pursue verification efforts that cannot be handled by the traditional model checking approaches. An additional benefit of our method is that in those cases where the property being analyzed does not hold we generate the execution paths that serve as counter-examples. We have implemented an automated tool-set that supports our approach and intend to expand both the tool-set and theory with heuristics for selecting compositional verification strategies and support for real-time properties.

References

[1] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. *Proceedings of the Workshop on Theory of Hybrid Systems*, 1992.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. H. Hwang. Symbolic model checking: 10^{20} states and beyond. *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, 428–439, 1990.

[3] M. Chiodo, T. R. Shiple, A. Sangiovanni-Vincentelli, and R. K.Brayton. Automatic reduction in CTL compositional model checking. *Proceedings CAV'92, LNCS 663*, 234–247, June 1992.

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

[5] S. C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. *Proceedings FSE'93, ACM SIGSOFT*, 115–125, December 1993.

[6] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.

[7] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems, LNCS 407*, 24–37, 1990.

[8] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, 464–475, June 1989.

[9] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[10] E. A. Emerson and C. Lei. Efficient model checking in fragments of the propositional mu-calculus. *Proceedings of Symposium on Logic in Computer Science*, 267–278, 1986

[11] J. Fischer and R. Gerber. Compositional Model Checking of Ada Tasking Programs *Proceedings COMPASS'94*. 1994.

[12] N. Halbwachs. Delay analysis in synchronous programs *CAV93, LNCS 697*. 1993.

[13] R. J. van Glabbeek. The linear time - branching time spectrum. *CONCUR90, LNCS 458*, 1990.

[14] R. Gerber and I. Lee. A Layered Approach to Automating the Verification of Real-Time Systems. *IEEE Trans. on Software Eng.*, 18(9):768–784, 1992.

[15] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. 1984.

[16] J. S. Ostroff. *Survey of formal methods for the specification and design of real-time systems*. Draft for IEEE Press book "Tutorial on Specification of Time", 1992.

[17] C. Stirling and D. Walker. CCS, liveness, and local model checking in the linear time mu-calculus. *Automatic Verification Methods for Finite State Systems, LNCS 407*, 1989.

[18] W. J. Yeh and M. Young. Compositional Reachability Analysis Using Process Algebra. *Proceedings TAV4, ACM SIGSOFT*, 49–59, October 1991.