A Survey on Compositional Verification

Hong Peng and Sofiène Tahar

Technical Report

Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3G 1M8 Canada
E-mail: {pengh,tahar}@ece.concordia.ca

November 1998

Abstract

In this survey, we overview several compositional verification approaches based on model checking, namely compositional minimization, classic assume guarantee reasoning and circular reasoning. We describe details of each of these approaches, and elaborate the general problems with some proposed solutions. We also summarize the advantages and disadvantages of each. In the last section, we give a general conclusion for future directions of compositional verification. Through the analysis, we discuss how the notions of preorder and abstraction play an important role in the reasoning[1]. Finally, we will see some new trends in the development of compositional verification methodologies, e.g., the integration of theorem proving and model checking approaches in the compositional reasoning context.

[1] Compositional reasoning is the core technique of compositional verification. In this paper, compositional reasoning will be used in some places to mean compositional verification.

1 Introduction

Many finite state systems are composed of multiple processes running in parallel. The specifications for such systems can often be decomposed into properties that describe the behavior of small parts of the system. An obvious strategy is to check each of the local properties using only the part of the system that it describes. If we can deduce that the system satisfies each local property, and if we know that the conjunction of the local properties implies the overall specification, then we can conclude that the complete system satisfies this specification as well.

This means that we separate the specification of a system into properties of its components and verify the properties of the components separately. This of course leaves us the obligation of proving that the component specifications in turn imply the specification of the entire system. This is called compositional verification.

In the simplest case, imagine we have a system composed of two processes P and Q, which communicate with each other over a channel or channels, and also communicate with their environment (See Figure 1)

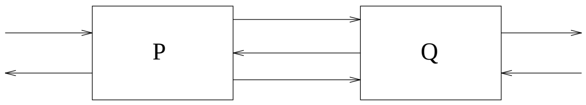

Figure 1: Two reactive machines

We denote this system $P||Q$, for the parallel composition of P and Q. Given component specifications φ_P for process P and φ_Q for process Q, we would like to reason according to the following schema:

$$\frac{\begin{array}{r} P \models \varphi_P \\ Q \models \varphi_Q \\ \varphi_P, \varphi_Q \models \varphi \end{array}}{P \parallel Q \models \varphi}$$

Here, $P \models \varphi$ means a model P satisfies a property φ. There are a number of difficulties involved in developing a verifier that can support this style of reasoning. First, we must be able to check whether every system containing a given component satisfies a given local property. Since it is often the case that the local property is only true under certain conditions, we need to be able to make assumptions about the environment of the component when doing the verification. These assumptions, which represent requirements on other components, must also be checked in order to complete the verification. In addition, we must provide a method for checking that the conjunction of certain local properties implies a given specification. This is called the "environment problem" [16]. Two possible solutions are considered: "compositional minimization" and "assume-guarantee reasoning".

Using compositional minimization technique [8][7][9][14], a "reduced" version Q' of Q is derived that characterizes just the behavior of Q that is visible to P via the channels connecting them (See Figure 2)

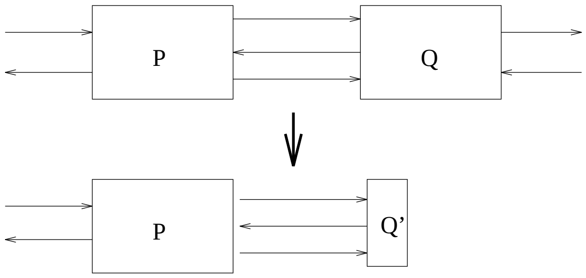

Figure 2: Compositional minimization

This reduced environment Q' has been called an "interface process" [8]. Hiding the communications of Q that are not visible to P may allow us to reduce Q to an interface process Q' with a very small number of states. The interface process Q' may be obtained, for example, by automatically minimizing the finite automaton representing Q. We will discuss this approach in Section 2.

In the assume-guarantee reasoning case [10][3][4][11][15][7][14][17][12], the environment is expressed by an assumption that process P makes on process Q using a temporal formula ψ_Q. These assumptions can be used when checking that P satisfies its specification φ_P, and must, of course, be discharged relative to Q. To do this, a reasoning rule is used such as the following:

$$\frac{\begin{array}{r} Q \models \psi_Q \\ P \parallel \psi_Q \models \varphi_P \end{array}}{P \parallel Q \models \varphi_P}$$

where $P \parallel \psi_Q$ means $P \parallel Q'$ with $Q' \models \psi_Q$. Here, the environment problem is just deferred but not solved because we have to show $Q \models \psi_Q$ without any environment. In addition, another difficulty is involved. That is how to get the assumption automatically.

The rest of this survey is organized as follows. Section 2 describes the compositional minimization approach. Section 3 describes the assume-guarantee reasoning approach. Section 4 presents another kind of assume-guarantee reasoning approach which allows circularity reasoning. Section 5 finally concludes the paper.

2 Compositional model checking

In the compositional minimization scheme [8][7][14], the environment of a process is modeled by another process called an interface process. A rule of interface called the interface rule provided the basis for the compositional model checking technique. The reader might think of the processes as being connected by a set of interconnecting as shown in Figure 1, where the interconnectings correspond to symbols in $\Sigma P_1 \cap \Sigma P_2$[2]. A_1 and A_2 are interface processes for P_2 and P_1, respectively. Intuitively, A_1 is all that P_2 can observe of P_1 through the wires that connect them. An analogous relationship holds between processes A_2 and P_1. Let φ and ψ be two formulas in a temporal logic L, two basic inference rules are proposed as follows.

$$\frac{\begin{array}{r} P_1 \downarrow \Sigma_{P_2} \equiv A_1 \\ \varphi \in L(\Sigma_{P_2}) \\ A_1 \parallel P_2 \models \varphi \end{array}}{P_1 \parallel P_2 \models \varphi}$$

$$\frac{\begin{array}{r} P_2 \downarrow \Sigma_{P_1} \equiv A_2 \\ \psi \in L(\Sigma_{P_1}) \\ P_1 \parallel A_2 \models \psi \end{array}}{P_1 \parallel P_2 \models \psi}$$

where $P_1 \downarrow \Sigma_{P_2} \equiv A_1$ means A_1 is a reduction of P_1 at the interface of P_2. If a minimization (reduction) procedure is known for P and L, then A_1 can be obtained by running this algorithm on $P_1 \downarrow \Sigma_{P_2}$. If such a procedure is not available, the rule will still be useful as long as the size of A_1 is significantly less than the size of P_1. A similar comment applies to A_2. The soundness of the above interface rule depends on the properties that are enumerated below[3]:

1. **Congruence:**
   $[(P \equiv P') \wedge (Q \equiv Q')] \rightarrow [(P \parallel Q) \equiv (P' \parallel Q')]$

2. **Distributivity of $\downarrow$ over $\parallel$:**
   If $\Sigma \supseteq \Sigma_P$, we must have

$$P \parallel (Q \downarrow \Sigma_P) \equiv (P \parallel Q) \downarrow \Sigma_P$$

3. **Property preservation 1**
   Equivalent machines satisfy the same formulas in the logic.

[2] Σ means a set of atom propositions of a logic or a formula

[3] Here, $\supseteq$ means subset; $\leftrightarrow$ means logical equivalence; $\equiv$ means semantic equivalence.

If $\Sigma_{P_1} \equiv \Sigma_{P_2}$, we must have

$$P_1 \equiv P_2 \rightarrow \forall \varphi \in L(\Sigma_{P_1})[P_1 \models \varphi \leftrightarrow P_2 \models \varphi]$$

4. **Property preservation 2**
   If $\varphi \in L(\Sigma)$ and $\Sigma \supseteq \Sigma_P$, we must have

$$P \downarrow \Sigma_P \models \varphi \leftrightarrow P \models \varphi$$

Given above four properties, we can deduce $P \parallel Q \models \varphi$ by assuming that $Q' \equiv Q \downarrow \Sigma_P$ (assumption 1) and $P \parallel Q' \models \varphi$ (assumption 2), i.e.,

$$\frac{\begin{array}{c} Q' \equiv Q \downarrow \Sigma_P \\ P \parallel Q' \models \psi \end{array}}{P \parallel Q \models \psi}$$

Proof

(1) With $P \equiv P$ and $Q' \equiv Q \downarrow \Sigma_P$ [assumption1], we can get
$P \parallel Q' \equiv P \parallel (Q \downarrow \Sigma_P)$ [Congruence];

(2) From (1) and using distributivity, we obtain $P \parallel Q' \equiv (P \parallel Q) \downarrow \Sigma_P$;

(3) With (2) and $P \parallel Q' \models \varphi$, we can get
$(P \parallel Q) \downarrow \Sigma_P \models \varphi$ [property preservation 1];

(4) Using (3) and property preservation 2, we finally obtain $P \parallel Q \models \varphi$. ∎

Above proof shows the soundness of the interface rules. Using this reasoning, we can handle arbitrary boolean combinations of formulas in an analogous manner, such as:

$$[(P_1 \parallel P_2) \models \varphi] \wedge [(P_1 \parallel P_2) \models \psi] \rightarrow (P_1 \parallel P_2) \models (\varphi \wedge \psi)$$

Now, we can make a sketch of this compositional reasoning. First, define a structure to describe a relation transition system. Second, this structure should have $\|, \downarrow, \equiv, \models$ operators over it and these operators should have above four properties. Third, a minimization algorithm $A_1 \equiv P_1 \downarrow \Sigma p_2$ should be designed. Following, we will use a Moore machine to illustrate this approach. Details of this example can be found in [8] [14] and [16].

Here, a structure is used to represent a Moore machine M, denoted as $struct(M)$. The composed Moore machine $M_1 \| M_2$ is denoted as $struct(M_1 \| M_2)$. φ is a CTL formula. if M is a closed Moore machine, then $M \models \varphi$ iff $struct(M) \models \phi$. If M is not closed, then $M \models \varphi$ when every closing environment M' for M, $struct(M \| M') \models \varphi$. In this way, to prove a Moore machine satisfying a property is transferred to prove a structure satisfying this property. The soundness of this definition is proved in [8].

To compute $P \downarrow \Sigma$, let's assume a small Moore machine (see Figure 3:(A)).

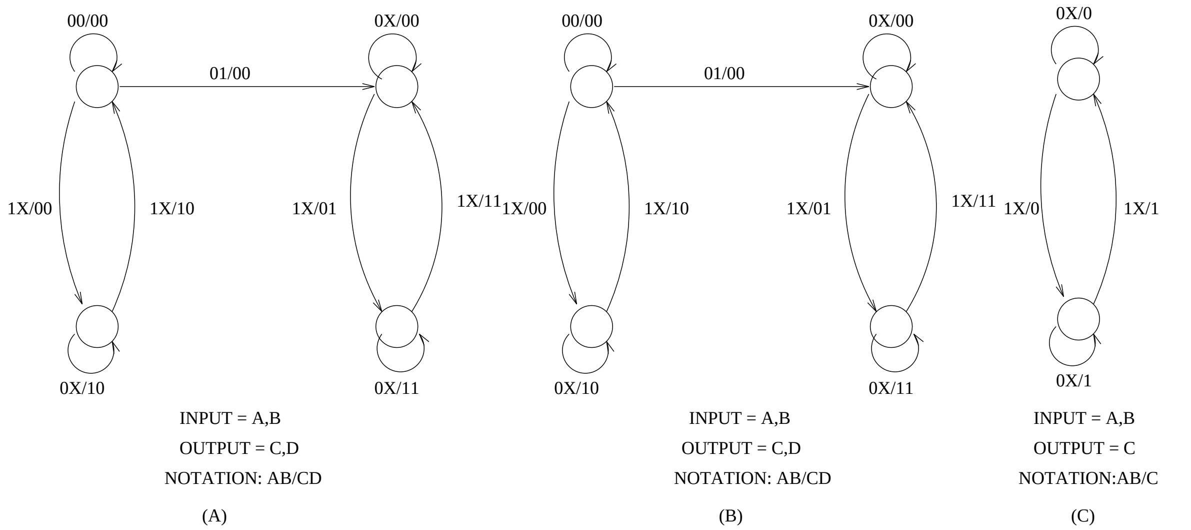

Figure 3: The compositional minimization of a small Moore machine

To compute $P \downarrow \{C\}$, we simply erase those outputs not in $\{C\}$. This gives Figure 3: (B).

Given this machine, the equivalence of Moore machines can be computed using the coarsest partitioning algorithm of [13]. Accordingly, the above machine can be minimized as illustrated in Figure 3: (C).

The advantage of above reasoning is the fact that it can be completely automatic. For example, this reasoning is implemented in SMV [9]. Clarke, et.al. [9] describe an application of the interface rule to reduce the complexity of verifying a system of communicating Moore machines . The example is the modular controller of a simple CPU with decoupled access and execution units. Both the access unit (AU) controller and the execution unit (EU) controller are deterministic Moore machines, described in the simple procedural state machine language called SML. By hiding some outputs of the EU controller that controls the EU datapath and applying the Hopcroft minimization algorithm [13] to produce a reduced interface process for verifying the AU controller, the number of states in the composition is reduced by roughly a factor of 6 in verifying the specification of the other module. The number of reachable states of $AU \| EU$ is 1274, while that of $AU \| EU'$ is 221.

The disadvantage of above compositional reasoning, although not mentioned in papers before 1994, is serious too. This is because the machine M by itself will be unlikely to satisfy φ, or any other useful property in an arbitrary environment. We have to be able to distinguish among different environments M'. That is, for a non-closed Moore machine, $M \models \varphi$ iff for every closing environment M$'$ for M, struct$(M \| M') \models \varphi$. Obviously, this is to prove φ being a tautology of M. The problem is that there is *probably* no efficient algorithm[4] to prove this [14], if φ is a CTL formula.

Although no mechanical proof can be given, we can image the reason. First, it is generally not possible to decompose a formula into subformulas, check the subformulas, and combine the results. For example, consider checking $\mathbf{EX}(\alpha = 1) \vee \mathbf{EX}(\alpha = 0)$ on a Moore machine where α is an input ranging over {0,1}. Obviously, the formula as a whole will be true regardless of what the environment does. However, $\mathbf{EX}(\alpha = 1)$ is certainly not true for all environments, nor is $\mathbf{EX}(\alpha = 0)$. Thus, determining whether the two subformulas are true in all environments does not help us solve the overall problem.

Second, because of the complexity of models under verification, we sometimes cannot just look at immediate successors when evaluating temporal formulas . Consider the Moore machine in Figure 4 [14], where we try to determine whether EXEX b = 1 is true for all systems containing the Moore machine shown in the figure. In standard CTL model checking, we would use the truth value for EX b = 1 at the two successors of the initial states to determine whether EXEX b = 1 was true at the initial state. For this example, there are environments that make EX b = 1 false at the left successor and others that make the formula false at the right successor. However, the overall formula is in fact true in all environments. This is because no environment can distinguish between the two successors based on their labeling. Hence, if the environment supplies the input a = 1 to the left successor, b = 1 becomes true in the next state. If it supplies only a = 0, then it must also supply a = 0 to the right successor, and this will again lead to a state where b is true. Thus we cannot just look at immediate successors when evaluating temporal operators.

In [14], D. E. Long and O. Grumberg have obtained some negative results which indicate that it is probably impossible to develop a fully general system of inference rules that will handle arbitrary temporal properties. In this case, the decomposed specifications cannot be proved to imply overall specification. It is a big disadvantage of this approach. Because of this reason, generally, a temporal logic without E-path quantifiers, ACTL, is used instead of CTL. ACTL is a subset of CTL containing formulas which have no E-path quantifiers when put in positive normal form. In compositional verification, ACTL is used instead of CTL.

[4] Here, efficient algorithm means this algorithm is exponential in input state variables, but polynomial in states and transition relations etc., because we cannot find an algorithm with a run time subexponential with respect to input state variables.

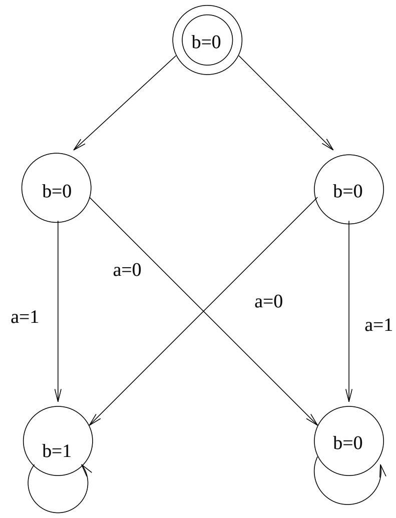

Figure 4: A nondeterministic Moore machine

3 Assume-guarantee in modular verification

The assume-guarantee style of verification was first advocated in the context of temporal logic by Pnueli [18]. In Pnueli's system, we work with triples of the form $\langle\varphi\rangle M\langle\psi\rangle$. The most common reading of such a triple is "if the environment of M satisfies φ, then M in this environment satisfies ψ" A typical chain of reasoning would be as follows:

$$\frac{\begin{array}{c}\langle\rangle M\langle\varphi\rangle\\ \langle\varphi\rangle M'\langle\psi\rangle\end{array}}{\langle\rangle M\|M'\langle\psi\rangle}$$

Here, we are asserting that if:

1. M satisfies φ; and
2. If the environment of M' satisfies φ, then M' satisfies ψ
3. Then the composition of M and M' will satisfy ψ.

The advantage of doing the verification in this manner is that, first, we never have to examine the composite state space of $M\|M'$. Instead, we check φ using just M, and then check ψ using only M' and the (hopefully simple) assumption φ. Second, this kind of chain reasoning gives us the possibility of doing hierarchical verification. The disadvantage is that the user must determine an appropriate φ. Knowledge of how the system should behave plus feedback from an automatic verifier makes this feasible in practice. More generally, we may use multiple levels of assumptions and guarantees when doing a verification. That is, once we have proved a guarantee, we may use that guarantee as an assumption in later stages. Because of this, a somewhat more precise reading of $\langle\varphi\rangle M\langle\psi\rangle$ would be "if the system satisfies φ and contains M, then the system also satisfies ψ." This is because φ

may in fact be something that is derived based on earlier assumptions about M, and may reflect these assumptions. Also, ψ may describe the combination of M and its environment, instead of just M. Of course, in order to avoid erroneous conclusions, all chains of deduction must be well founded, i.e., the base assumptions must themselves be proved without any assumptions, like $\langle\rangle M\langle\varphi\rangle$. As we already mentioned in last section, it is hard to prove a formula of a component without specific environment if we use a full set of CTL. An efficient solution might be again ACTL.

However, the key issue in assume-guarantee reasoning is to establish the truth of a triple $\langle\varphi\rangle M'\langle\psi\rangle$. An elegant way to obtain a system with the above properties is to provide a preorder[5] on the finite state models that captures the notion of "more behaviors" and to use a logic whose semantics relate to the preorder. The preorder should preserve satisfaction of formulas of the logic, i.e., if a formula is true for a model, it should also be true for any model which is smaller in the preorder:

$$(P \preceq Q) \wedge P \models \varphi \rightarrow \ P \models \varphi$$

Another requirement for the reasoning is that composition should preserve the preorder. That is,

$$(P \preceq P') \wedge (Q \preceq Q') \rightarrow \ (P\|Q) \preceq (P'\|Q')$$

In additional, following condition is needed in the assume-guarantee reasoning.

$$\text{for all formulas } \varphi \text{ w.r.t. P there exists a process } T(\varphi) \text{ such that}$$
$$P \preceq T(\varphi) \textit{ iff } P \models \varphi$$

In such a framework, the soundness of the assume-guarantee reasoning

$$\frac{\begin{array}{c}\langle\rangle M\langle\varphi\rangle \\ \langle\varphi\rangle M'\langle\psi\rangle\end{array}}{\langle\rangle M\|M'\langle\psi\rangle}$$

can be proved as:

Proof

$\langle\rangle M\langle\varphi\rangle \rightarrow M \preceq T(\varphi)$;

$M' \preceq M'$ (reflexive);

$M'\|M \preceq M'\|T(\varphi)$;

$\langle\varphi\rangle M'\langle\psi\rangle \rightarrow M'\|T(\varphi) \preceq T(\psi)$;

$M\|M' \preceq T(\psi)$ (transitive);

$\langle\rangle M\|M'\langle\psi\rangle$ ∎

[5] A preorder relation $\preceq$ is a reflexive and transitive relation. It satisfies "$P \preceq P$" and "$P \preceq Q \wedge Q \preceq R \rightarrow P \preceq R$".

In [10], a notion of "*simulation relation* $\preceq$" between state transition systems is introduced as the preorder. We could in fact view this preorder relation as the basic relationship between an implementation and a specification. $P \preceq Q$ means "P refines Q" or "Q simulates P" or "Q abstracts P".

Because of the transitivity of $\preceq$, we would get hierarchical verification essentially for free. For example, if $M \preceq M'$ ("M' can simulate M"), and if we want to know whether $M \preceq M''$, then it would be enough to check that $M' \preceq M''$. Here, M' would represent a specification of M that is used to prove a higher level specification M''. The simulation relation will also interact with composition in a nice way: if $M \preceq M'$, then we will have $M \| M'' \preceq M' \| M''$. This type of property allows us to replace an implementation by its specification in a composition.

Another key point in this assume-guarantee reasoning is that any ACTL formula ψ has a maximal model $T(\psi)$ which can be obtained by a "tableau construction". This tableau construction maps a formula φ to an associated state transition system $T(\varphi)$ which is called the tableau of the formula. The standard model checking algorithm is used at one level, then we construct tableaus for the specification formulas and use them as implementations at the next level.

In above framework, the classic assume-guarantee reasoning can be defined as:

$$\frac{\begin{array}{c} M \preceq T(\varphi) \\ M' \| T(\varphi) \preceq T(\psi) \end{array}}{M \| M' \preceq T(\psi)}$$

As a final note, this reasoning will actually be working with structures rather than Moore machines. This is mainly because formulas do not have notions of inputs and outputs, so the tableau construction will most naturally produce structures. In addition, structures can serve as "intermediate language" for representing other, more complex types of models.

Following, we use an example to illustrate above approach [14]. Let the structures for two circuits be denoted by M and M'. We use $AG(r = 0 \rightarrow AXq = 0)$ as an assumption on M', and then prove the desired property by combining this assumption with M:

$$\frac{\begin{array}{c} \langle\rangle M' \langle AG(r = 0 \rightarrow AXq = 0)\rangle \\ \langle AG(r = 0 \rightarrow AXq = 0)\rangle M \langle AG(p = 0 \vee q = 0)\rangle \end{array}}{\langle\rangle M \| M' \langle AG(p = 0 \vee q = 0)\rangle}$$

Checking $\langle\rangle M' \langle AG(r = 0 \rightarrow AXq = 0)\rangle$ will be done with the standard model checking techniques. However, in order to check

$$\langle AG(r = 0 \rightarrow AXq = 0)\rangle M \langle AG(p = 0 \vee q = 0)\rangle$$

we need to construct the tableau for $AG(p = 0 \vee q = 0)$ and compose it with M. The states of the tableau will have valuations for r and q, plus information about the elementary formulas (For details, see [14]). After constructing the tableau and performing the model checking, we find that

$$\langle AG(r = 0 \rightarrow AXq = 0)\rangle M \langle AG(p = 0 \vee q = 0)\rangle$$

does hold. We hence conclude

$$\langle\rangle M \| M' \langle AG(p = 0 \vee q = 0)\rangle$$

In above discussion, we illustrated a number of advantages of the assume-guarantee reasoning. However, it does have some disadvantages.

As we mentioned in the previous section, CTL cannot be handled in current compositional verification including the assume-guarantee reasoning. Can the full set of CTL be handled? In [8], only boolean combination of temporal properties can be handled. In [10], although not strictly proved, CTL possibly cannot be handled in current frameworks. ACTL has to be used instead of CTL. In practice, there are actually some cases which cannot be expressed by ACTL, such as $\mathbf{AGEG}\neg\varphi$ (φ is a CTL property). So, either we can handle CTL in a compositional framework or we have to find another expressive temporal logic, like ATL [4]. In addition, as we know before, the reason of this problem is the so called "environment problem". We have to prove a tautology of a φ. Actually, every component will be working under a specific environment [6]. It seems unnecessary to prove a tautology of a temporal logical formula φ. So, can we only prove that φ holding under a specific environment? In the assume-guarantee context, this question can be translated to "Can we use assumption at the very first stage of the reasoning?" or "does there exist another elaborative form of assume-guarantee reasoning?".

Regarding the hierarchical verification, the difficulty lies in finding a series of assumptions which can express different levels of environment. It is unlikely that a tool can provide such assumptions automatically. However, it is very possible to have a semi-automatic solution in terms of hints to the user [11].

Since the solutions of above problems are still open, in next section, we will introduce a circular reasoning which can avoid the "environment problem".

4 Reactive modules

In the work of [1] and [2], environment assumptions are expressed in a circular manner. Here, we present a recent circular reasoning called "reactive modules".

The notion of reactive modules is a formalism for the modular description of systems with heterogeneous components. The definition of reactive modules can be found in [3]. The target application is hardware/software codesign and verification. This model formalizes heterogeneous systems that are built from synchronous and asynchronous hardware and software components, and provides circular assume-guarantee and abstraction principles for reasoning about such systems. Here we give only a brief introduction.

The state of a reactive module is determined by the values of three kinds of variables [3]: the *external variables* are updated by the environment and can be read by the module; the *interface variables* are updated by the module and can be read by the environment; the *private variables* are updated by the module and cannot be read by the environment. The external and interface variables are called *observable variables*; the interface and private variables, *controlled variables*.

The state of a reactive module changes in a sequence of rounds. The first round is called the *initial round*, and determines initial values for all variables. Each subsequent round is called an *update round*, and determines new values for all variables. For external variables, the values in the initial and update rounds are left unspecified (i.e., chosen nondeterministically). For controlled variables, the values in the initial and update rounds are specified by (possibly nondeterministic) guarded commands. In each update round, the new value of a controlled variable may depend on the (latched) values of some variables from the previous round. In addition, in each round, the initial (or new) value of a controlled variable may depend on the initial (or new) values of some other variables from the same round; such a dependency between the values of variables within a single round is called an *await dependency*. In order to avoid inconsistent specifications, the await dependencies must be acyclic. In reactive modules, the acyclicity restriction is enforced statically, by partitioning the controlled variables into atoms that can be ordered such that in each round, the initial (or new) values for all variables of an atom can be determined simultaneously from the initial (or new) values of the external variables and the variables of earlier atoms. Each round, therefore, consists of several subrounds for the external variables, and one per atom. Each atom has an *initial command*, which specifies the possible initial values for the variables of the atom, and an *update command*, which specifies the possible new values for the variables of the atom within each update round. In the update command, unprimed occurrences of variables refer to the latched values from the previous round; in both the initial and update commands, primed occurrences of variables refer to the initial (or new) values from the same round.

The composition operation combines two reactive modules into a single module whose behavior captures the interaction between the two component modules. the composition $P\|Q$ is the module with the set $privX_{P\|Q} = privX_P \cup privX_Q$ of private variables, the set $intfX_{P\|Q} = intfX_P \cup intfX_Q$ of interface variables, the set $extlX_{P\|Q} = (extlX_P \cup extlX_Q) \backslash intfX_{P\|Q}$ of external variables, and the set $A_{P\|Q} = A_P \cup A_Q$ of atoms.

The module P implements the module Q, written P $\preceq$ Q, if the following conditions are met: (1) every interface variable of Q is an interface variable of P ; (2) every external variable of Q is an observable variable of P ; (3) for all observable variables x of Q and all interface variables y of Q, if y$\succ_Q$x, then y$\succ_P$x[6]; and (4) if $\bar{s}$ is a trace of P, then the projection $\bar{s}[obsX_Q]$ is a trace of Q[7].

The first three conditions ensure that the compatibility constraints imposed by P on its environment are at least as strong as those imposed by Q. The fourth condition is conventional trace containment. Intuitively, if P$\preceq$Q, then the module P is as detailed as the module Q: the implementation P has possibly more interface and external variables than the specification Q; some external variables of Q may be interface variables of P, and thus are more constrained in P; the implementation P has possibly more await dependencies among its observable variables than the specification Q; and P has possibly fewer traces than Q, and thus more constraints on its execution. It is easy to check that every module P implements itself, and that if a module P implements another module Q, which, in turn, implements a third module R, then P also implements R (i.e., reflexivity and transitivity).

The implementation relation $\preceq$ can be proved being a preorder relation [3].

Using this preorder operator, we can do following assume-guarantee reasoning: Let P_1 and P_2 be two compatible[8] modules, and let Q_1 and Q_2 be two compatible modules such that every external variable of $Q_1\|Q_2$ is an observable variable of $P_1\|P_2$. If $P_1\|Q_2 \preceq Q_1$ and $Q_1\|P_2 \preceq Q_2$, then $P_1\|P_2 \preceq Q_1\|Q_2$.

Briefly, we can express above proposition as:

$$\frac{\begin{array}{c} Q_1\|P_2 \Rightarrow Q_2 \\ Q_2\|P_1 \Rightarrow Q_1 \end{array}}{P_1\|P_2 \Rightarrow Q_1\|Q_2}$$

Here, we assume that module Q_2 satisfies some specification P_2 to prove that Q_1 satisfies P_2 to prove that Q_1 satisfies P_1, and vice versa. Generally, this apparent circularity is not sound as we mentioned in Section 2. However, this circularity can be broken by induction

[6] y$\succ$x means the variable y awaits the variable x

[7] Trace is an execution sequence of observable variables. In the newest paper of Henzinger et.al. [11], this trace concept is replaced by simulation.

[8] Two modules P and Q are compatible if (1) the controlled variables of P and Q are disjoint, and (2) the await dependencies between the variables of P and Q are acyclic.

over time. That is, let the notation $P\uparrow^{\tau}$ stand for $\forall t.P$ or "P holds up to time t = τ". We can soundly reason as follows, by induction on τ:

$$\frac{\begin{array}{c} P_1 \uparrow^{\tau-1} \Rightarrow P_2 \uparrow^{\tau} \\ P_2 \uparrow^{\tau} \Rightarrow P_1 \uparrow^{\tau} \end{array}}{\forall t.(P_1 \wedge P_2)}$$

In the base case, when $\tau = 0$, note that $P_1 \uparrow^{\tau-1}$ is a tautology. Hence, we have $P_2 \uparrow^{0}$, and thus $P_1 \uparrow^{0}$, $P_2 \uparrow^{1}$, $P_1 \uparrow^{1}$ and so on. By reasoning inductively, we use each process's specification as the environment of the other, avoiding circularity. In reactive modules, this induction refers to a result of the delay of one time unit from input to output of the machines.

In this reactive modules framework, a verification tool "MOCHA" is being currently developed at UC Berkeley [4][11]. In another approach, theorem proving is being involved in compositional verification [17]. It is something natural because the current scale of circuit can be beyond the capability of model checking easily.

5 Future Directions

While symbolic representations have greatly increased the size of the systems that can be verified, many real systems are still too large to be handled. Thus, it is important to find techniques that can be used in conjunction with the symbolic methods to extend the size of the systems that can be verified. The compositional "divide and conquer" method is the most classical one. In following, we will discuss the lessons we learned form the previous sections.

One of the ways to do compositional verification is to actually verify a smaller in size model, which is equivalent to the given one with respect to the temporal logic used for specification. Two models P_1 and P_2 are *equivalent* with respect to some temporal logic L, if for every formula φ in L, P_1 satisfies φ if and only if P_2 satisfies φ. This means that instead of checking if the given model P_1 satisfies the formula φ we can check if a smaller model P_2 satisfies φ. If P_2 satisfies φ so does P_1. If P_2 does not satisfy φ so does P_1. The equivalence requirement between structures is sometimes too strong. In such a case it is possible to use a preorder relation.

Given two models P_1 and P_2, P_1 is smaller by the preorder than P_2 ($P_1 \preceq P_2$) with respect to a temporal logic L, if for every formula φ in L, if P_2 satisfies φ then P_1 satisfies φ. As before, instead of checking if the given model P_1 satisfies a formula φ we will check if P_2 satisfies φ. P_2 must be smaller in size, but greater by the preorder than P_1. If P_2 satisfies φ so does P_1. Many approaches to state explosion problem, such as abstraction, modularity, symmetry and partial order are based on equivalences and preorders. This is also used in approaches in previous sections. Milner's strong simulation and weak simulation [18]has been and will be used in compositional verification more and more often.

The second lesson we learned from previous research work is that, to do compositional verification, model reduction only (approach in Section 1) is not enough. It has been commonly accepted that an efficient algorithm cannot be obtained for these approaches. In this case, the abstraction of the target system is necessary.

In this survey, we listed the earlier approaches (Section 1, before 1989) and the newer approaches (Sections 2 and 3). We can find a clear trend that more and more researchers are trying to integrate model checking and theorem proving. The reason is obviously that model checking is a finite state solution. State space explosion can easily be reached in verifying real circuits, especially, when software interfaces or software itself is considered. In the compositional verification context, no matter what approaches are used, the reasoning itself poses a problem which is beyond the capability of model checking, such as the "environment problem" or induction over time. So, it is quite natural to integrate theorem proving into compositional model checking. As a matter of fact, we can find some pilot work on this topic [15][17][5]. However, this is still a very open topic.

Through the analysis of the above sections, we know that there is a long way to go if we want to put compositional verification into practice, making this area a promising research topic.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[3] R. Alur and T. Henzinger. Reactive modules. In *Proc. 11th IEEE Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[4] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: modularity in model checking. In A.J. Hu and M.Y. Vardi, editors, *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science 1427, pages 521–525. Springer-Verlag, 1998.

[5] K. Bhargavan, C. A. Gunter, E. L. Gunter, and M. Jackson. The village telephone system: a case study in formal software engineering. In *Proc. of TPHOL'98: Theorem Proving in HOL*, 1998.

[6] E. Chang. *compositional verification of reactive and real-time systems.* PhD thesis, Standford U., 1995.

[7] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking. In *Nato ASI*, volume 152 of *F.* SpringerVerlag, 1996.

[8] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. 4th Annual Symposium on Logic in Computer Science*, pages 353–62, Pacific Grove, CA, June 1989.

[9] E. M. Clarke, D. E. Long, and K. L. Mcmillan. A language for compositional specification and verification of finite state hardware controllers. In *Proc. of Hardware Description Languages*, 1989.

[10] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[11] T. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. Computer Aided Verification (CAV'98)*, Vancouver, BC, Canada, June/July 1998.

[12] T. Henzinger, S. Qadeer, S. K. Rajamani, and S.Tasiran. An assume guarantee rule for checking simulation. In G. Gopalakrishnan and P. Windley, editors, *FMCAD 98: Formal Methods in Computer-aided Design*, volume 1522 of *Lecture Notes in Computer Science 1522*, pages 421–432. Springer-Verlag, 1998.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory,Languages, and Computation.* AddisonWesley, 1979.

[14] D. E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, CMU, 1993.

[15] K. L. McMillan. Compositional rule for hardware design refinement. In *Proc. Computer Aided Verification (CAV'97)*, pages 24–35, Haifa Israel, June 1997.

[16] K. L. McMillan. Tutorial on model checking, 1998. http://www-cad.eecs.berkely.edu/kenmcmil/.

[17] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proc. Computer Aided Verification (CAV'98)*, Vancouver, BC, Canada, June/July 1998.

[18] A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series. Series F*. SpringerVerlag, 1984.