Compositional Verification of Component-based System

Wang Qiang
RiSD, I&C, EPFL

Abstract—Ensuring safety properties of the system model is an essential requirement of rigorous system design. Formal verification provides a rigorous way to prove or disprove the safety properties of system model with respect to certain formal specifications. While as the rapid growth of the complexity of systems, traditional automatic verification approaches can hardly scale. The objective of this research is to develop the theory, technique and tool support for automatic compositional verification of component-based systems.

Index Terms—Compositional Verification, BIP, Abstract Interpretation, Model Checking, Speculative Linearizability

I. INTRODUCTION

RIGOROUS system design [1] requires the use of a single powerful component framework providing primitives for building complex systems by assembling simpler components. It allows us to develop large scale systems in a divide-and-conquer manner and mitigate the complexity of system design by offering incrementality in the construction phase.

BIP (Behaviour, Interaction, Priority) [1] [2] is a general framework encompassing component-based system design. It relies on the separation between computation and coordination.

In BIP, complex systems are constructed by composing a set of atomic components. The behaviour of each atomic component is described as a state transition system, while the behaviour coordination between components is structured into two layers: interaction layer and priority layer. Interaction specifies the synchronisation between components, while priority provides a mechanism to schedule the interactions when several interactions are enabled.

Motivated by pragmatic reasons, two kinds of safety properties are of essential interests in BIP systems: invariants and deadlock-freedom. Invariants are state predicates preserved by the transition relation, that is if an invariant holds at some state, then it holds at all its successor states. Deadlock-freedom means that at least one transition is enabled from any reachable states. It is important to guarantee that the system model respects these safety properties as early as possible in the system design flow. In general, there are two methods to achieve this goal: theorem proving and model checking. However, theorem proving is difficult to be automated, which prevents it from being used in large scale component-based systems. In this study I will focus on model checking techniques for safety properties, in particular the techniques for invariant verification. Notice that deadlock-freedom verification can be transformed into the invariant verification.

Model checking [6] is a fully automatic verification technique to check whether a system model meets the specification of property or not. Usually the system is modelled as a finite state machine and the specification is expressed as a formula in the temporal logic. Then the verification is performed by using an exhaustive search procedure on the state space graph of the system model. If the property is satisfied by the system model, model checking can prove it. While if the property does not hold, model checking can provide a counterexample, which shows why the property specification is violated. Model checking has been proved a success in hardware design. However, for component-based systems, model checking is still difficult to scale due to the following reasons:

1) Components are usually infinite-state systems, which makes exhaustive search impossible;
2) The inherent concurrency in component-based systems makes the state space blow up, as a result the model checking is computationally infeasible.

The objective of this research is to develop the theory, technique and tool support for verifying essential safety properties of component-based systems, particularly the BIP systems in an automatic and scalable manner.
II. REVIEW OF THE SELECTED PAPERS

In this section, I will present the review of techniques presented in selected papers [5] [4] [13] essential for achieving the goals of this proposal. The flow of this proposal is not structured by these papers, but in a logic way based on the relate techniques. First I introduce some basic conceptions that will be used throughout this proposal.

The unified mathematical model for computing systems, as well as programs is defined by a transition system.

**Definition (Transition System)** A transition system (TS) is a tuple \((S, R, I)\) where:

1. \(S\) is a finite set of states;
2. \(R \subseteq S \times \text{Act} \times S\) is a set of transition relations;
3. \(I \subseteq S\) is the set of initial states.

Given a transition system \(TS = (S, R, I)\), the associated post and pre predicate transformers \([9]\) are denoted as follows for a predicate \(P\):

\[
\text{post}(P) = \{ s' \in S | \exists s \in S, P(s) \land (s, s') \in R \}\]

\[
\text{pre}(P) = \{ s' \in S | \exists s \in S, P(s) \land (s', s) \in R \}
\]

Similarly we define \(\text{pre}(P) = \neg \text{pre}(\neg P)\) to be the set of states from which it is not possible to reach \(\neg P\) by activating one transition in \(TS\):

\[
\text{pre}(P) = \{ s' \in S | \forall s \in S (s', s) \in R \Rightarrow P(s) \} \}
\]

The same extension can be defined for \(\text{post}(P)\).

A trace of a transition system is defined by a sequence of states and the states that can be reached from these initial states

**Definition (Invariant)** Let \(TS = (S, R, I)\) be a transition system. Predicate \(P\) is called an invariant of \(TS\) if \(\forall s \in S\) such that \(s\) is reachable from \(I\), then \(s\) satisfies \(P\), denoted by \(P(s)\). \(P\) is inductive if \(\forall s \in I, P(s)\) and \(\forall (s, s') \in R, P(s) \Rightarrow P(s')\). (An inductive invariant is trivially an invariant.)

Using predicate transformers, we have: \(P\) is an inductive invariant iff

1. \(P \Rightarrow \text{pre}(P)\), or \(\text{post}(P) \Rightarrow P,\) or
2. \(P = P \land \text{pre}(P),\) or \(P = P \lor \text{post}(P)\)

The strongest inductive invariant \(P\) can be expressed as the least fixed point of \(\text{post}\) operator. Since the transition system has finite set of states, the existence of least fix point can be guaranteed by the monotonic property of \(\text{post}\). More information can be found in [10]. This strongest inductive invariant characterizes exactly the set of reachable states from \(I\) which we denote by \(R(TS)\).

To check if a transition system \(TS\) satisfies an invariant \(P\), a naive approach is to exhaustively search all the possible reachable states \(R(TS)\) and check if \(P\) is satisfied on every state. If \(P\) is satisfied by every reachable state, we can conclude that \(TS\) is safe with respect to \(P\). Otherwise, we can find a state, which violates \(P\) and conduct a backward search from it to construct the counterexample.

A. Symbolic model checking

A well-known improvement of model checking algorithm is the symbolic model checking [7] [5]. The basic idea is that instead of doing exhaustive search on explicit state space, we can represent implicitly the state space, as well as the transition relations as boolean formulas and then perform all computations by manipulating these boolean formulae. In practice, boolean formulae are represented by Binary Decision Diagrams (BDD), which provides a canonical form for boolean formulae that is more compact than conjunctive and disjunctive normal form.

Assuming that the safety property specified with an invariant \(Q\) and the set of initial states \(I\) are both represented as BDDs, the rough procedure for checking \(Q\) would be to compute the reachable states by the iterations \(R(TS) = I \lor \text{post}(I) \lor \text{post}^2(I)\),... where operations \(\lor\) and \(\lor\) are both carried out in terms of BDD. At each step, the iteration computes the set of states that can be reached in \(i\) steps and intersects the new states with the set of states violating the property, which can be specified as \(\neg Q\). If the intersection is not empty, it means that an violation has been detected and a counterexample has been found. Otherwise we continue iterating until a fixed point is reached and the if the intersection is still empty, we can conclude the safety property \(Q\) is satisfied by the system model \(TS\).

Due to the symbolic representation of boolean formulae, this approach is suitable for verifying systems with a large set of states. However, symbolic model checking still has its limitations. Since the size of BDD is very sensitive to ordering of boolean variables, and it is in general very difficult to find a good ordering, in some cases the BDD representation can be exponential in the size of system description. This makes symbolic model checking computationally infeasible for large systems. Moreover, the computation \(\text{post}\) in terms of BDD is also very expensive especially when BDD becomes very large, so the fixed point iteration is infeasible in most cases. As we will see later, symbolic model checking can be combined with other abstraction techniques to improve the performance.

B. Bounded model checking

An alternative method that can reduce the high complexity of complete symbolic model checking is the bounded model checking [14] [5]. The basic idea of this method is to search for a counterexample within a bounded number of execution steps, rather than computing the fixed point of operator \(\text{post}\). Given a fixed steps of execution \(k\), bounded model checking constructs a boolean formula that represents the set of initial states and the states that can be reached from these initial states within \(k\) steps:

\[
R^k(TS) = I \lor \text{post}(I) \lor \text{post}^2(I)\ldots \lor \text{post}^k(I)
\]

Notice, that in this formula we do not have to compute the \(\text{post}\) explicitly but it can be obtained by unwinding the entire transition relations only up to \(k\) length. For loops in the program, an optimized unwinding strategy is to unwind the loops separately. Then the satisfiability of conjunction \(R^k(TS) \land \neg Q\) is checked, where \(Q\) is the property to be
checked. If the conjunction is satisfiable, bounded model checking provides a counterexample of length at most \( k \), which shows the violation of property \( Q \). If it is not satisfiable, we can either increase \( k \) hoping we can find a violation in the next steps or stop searching without reaching a conclusion. Because the absence of errors in \( k \) execution steps does not guarantee the absence of errors afterwards, so it does not prove the correctness by looking at the \( k \) execution steps. In this sense, bounded model checking is incomplete. To make it complete, a possible solution is to choose the longest path among the shortest paths between any two states (i.e., the maximum value \( k_{\text{max}} \) of \( k \)), also called the completeness threshold, which ensures that states that can be reached in \( k_{\text{max}} \) steps cover all the reachable states. However, finding the \( k_{\text{max}} \) is extremely hard.

Bounded model checking reduces the safety property verification problem to a propositional satisfiability (SAT) problem that can be efficiently solved by modern SAT solvers. However, the complexity of satisfiability checking grows rapidly with the increase of \( k \). This makes bounded model checking infeasible for large component-based systems.

C. Abstract static analysis

Unlike the symbolic model checking or bounded model checking techniques, where the analysis, which is carried out over a relatively large property domain is too expensive to reach a conclusion in most cases, abstract static analysis tries to avoid this complexity by designing a simpler abstract property domain, in which the analysis is guaranteed to be computationally cheaper and, therefore, relating the abstract analysis to the concrete one. Originally, abstract static analysis encompasses a family of techniques for automatically computing approximate information about the behaviour of a program without executing it [10] [5]. However, since most questions about the behaviour of a program are either undecidable or computationally infeasible to answer, the essence of abstract static analysis is to efficiently compute sound approximations of the program behaviour in the sense that the program being proved safe in the abstract analysis is indeed safe, but the abstract analysis may produce spurious behaviours due to the approximations. In practice, the approximations are chosen to offer the best trade-off between the precision and the efficiency of analysis.

Static program analysis can be formalized by the abstract interpretation framework which relates the analysis over an abstract domain and the analysis over a concrete domain [10]. The correspondence between the concrete and abstract domains can be established by a Galois connection.

Definition (Galois connection) Let \((D^p, \sqsubseteq)\) and \((D^a, \sqsubseteq)\) be the concrete and abstract property domain respectively, a pair of (monotone) functions \((\alpha, \gamma)\) forms a Galois connection between these two domains, where \(\alpha : D^p \rightarrow D^a\) and \(\gamma : D^a \rightarrow D^p\), iff for all \(a \in D^p\) and \(b \in D^a\) holds:

\[
\alpha(a) \sqsubseteq b \iff a \sqsubseteq \gamma(b),
\]

where \(\alpha\) is often called abstraction function, and \(\gamma\) is often called the concretisation function.

Definition (Function abstraction) Given a concrete domain \((D^p, \sqsubseteq)\), an abstract domain \((D^a, \sqsubseteq)\) and a Galois connection \((\alpha, \gamma)\) between them, a transfer function \(F^\sharp \in D^p \rightarrow D^a\) is the approximation of transfer function \(F^\natural \in D^p \rightarrow D^a\) iff

\[
\alpha \circ F^\sharp \circ \gamma \subseteq F^\natural.
\]

\(F^\natural\) is the exact function abstraction of \(F^\sharp\) when \(F^\natural = \alpha \circ F^\sharp \circ \gamma\) (\(\circ\) denotes functional composition). In this case, we also say \(F^\natural\) is the induced abstraction of \(F^\sharp\) by Galois connection \((\alpha, \gamma)\).

The abstract static analysis consists in computing an approximation of a program semantics expressing the properties of interest of the program \(p\) to be analysed. The semantics can often be expressed as a least fixed-point \(S^\sharp[p] = lfp^p F^\sharp[p]\) that is the least solution to a monotonic system of equations \(X = F^\natural[p](X)\) computed on a concrete domain \((D^p, \sqsubseteq)\) where the transfer function \(F^\natural[p] \in D^p \rightarrow D^a\) is monotonic. The approximation is formalized through a Galois connection \((\alpha, \gamma)\) from concrete domain \((D^p, \sqsubseteq)\) to an abstract domain \((D^a, \sqsubseteq)\), where a concrete program property \(p \in D^p\) is approximated by any abstract property \(\gamma(p')\in D^a\) such that \(p \subseteq \gamma(p')\) and has a best precise abstraction \(\alpha(p) \in D^a\). Then the abstract static program analysis consists in compute the abstract least fixed point \(S^\sharp[a] = lfp^a F^\sharp[a]\) which is a sound approximation of the concrete semantics in that \(lfp^a F^\sharp[a] \subseteq \gamma(lfp^p F^\natural[p])\). This fixed point soundness condition can be ensured by stronger functional soundness conditions such as \(F^\sharp[a]\) is monotonic and it is the function abstraction of \(F^\natural\) (i.e., \(\alpha \circ F^\sharp[a] \circ \gamma \subseteq F^\natural[a]\)). Widening and narrowing techniques can be used to accelerate the convergence of fixed points computation.

The precision of an abstract static analysis depends on the expressiveness of an abstract domain which is an approximate representation of concrete properties. In the past decades, various abstract domains have been designed for different purposes of analysis. Numerical abstract domains are used for computing invariants of numeric variables. The domain of Signs has three values: \{Positive, Negative, Zero\}. Intervals are more expressive as the values in the Signs domain are modelled by the intervals \([min, 0], [0, 0],\) and \([0, max]\). There are many others, such as the domain of Congruences, Octagons, and Polyhedra etc. These domains are useful for computing inequality constraints about integer variables and for deriving such invariant for programs without pointers. As for programs with pointers, storage shape graphs are often used in the shape analysis as the abstract domain.

Despite the efficiency, abstract static analysis usually yields lots of spurious alarms due to the approximations. Although simpler abstract domains and analyses may be useful for detecting program errors, such as arithmetic overflows and array bound exceeding, they rarely suffice for invariant verification for large programs or component-based systems. Besides unlike model checking, generating counterexamples is difficult or even impossible, due to the loss of precision in join and widening operations of the abstract analysis. There is no general way to make abstract static analysis complete. A particular case is the counterexample guided abstraction refinement approach based on predicate abstraction, which will
be discussed in the next section.

D. Predicate abstraction

The idea of predicate abstraction can be understood as a combination of model checking techniques and abstract static analysis. To check an invariant of a transition system, we can select a finite set of predicates which are relevant to the invariant and use them to approximate the concrete system states, and then build an abstract transition system by mapping each concrete state to an abstract one which is the valuation of the pre-selected predicates and by building abstract transition relations between abstract states. After that, we can apply model checking techniques to check if the invariant is satisfied by the abstract transition system. If it is satisfied, then we can conclude that it is also satisfied by the concrete transition system due to the soundness condition of predicate abstraction. However, when the abstract transition system does not satisfy the invariant and the model checking produces a counterexample, we can not conclude the concrete system violates the invariant because the counterexample may be resulted from the loss of precision due to the coarse abstraction. In this case, abstraction refinement is required to rule out the spurious counterexample by discovering and adding new predicates, where we continue to model check the refined abstract system until the invariant is proved or disproved by finding a real counterexample. This abstraction, verification and refinement paradigm is also called counterexample guided abstraction refinement (CEGAR) [8] [5].

Predicate abstraction can be formalized by using abstract interpretation framework. Let \( TS = (S, R, I) \) be a transition system, the concrete properties we are concerned with are sets of states, thus the concrete property domain is the power-set lattice \( D^S = (2^S, \subseteq) \) with subset inclusion ordering \( \subseteq \). Given a finite set of predicates: \( P = \{p_1, ..., p_n\} \), the abstract domain of predicate abstraction can be represented as the power-set of bitvectors (each bit corresponds to a predicate) of length \( n \), that is, \( D^\alpha = (2^{(0,1)^n}, \subseteq) \), where order \( \subseteq \) is the subset inclusion. The abstraction function is defined by:

\[
\alpha_\beta : 2^S \rightarrow 2^{(0,1)^n},
\]

\( \alpha_\beta(S) = \{v_1, ..., v_n\} | \exists s \in S \ s |= v_1 \cdot p_1 \land ... \land v_n \cdot p_n \}, \)

where \( 0 \cdot p_i = \neg p_i \) and \( 1 \cdot p_i = p_i \).

The concretization function is defined by:

\[
\gamma_\beta : 2^{(0,1)^n} \rightarrow 2^S,
\]

\( \gamma_\beta(V) = \{s | \exists v_1, ..., v_n \in V \ s |= v_1 \cdot p_1 \land ... \land v_n \cdot p_n \}, \)

Given \( D^\beta = 2^{(0,1)^n} \), and the abstraction function \( \alpha_\beta \), the best abstraction of the image operator \( post \) is the operator \( post_\beta \) on abstract domain defined by:

\[
post_\beta = \alpha_\beta \circ post \circ \gamma_\beta,
\]

Hence, the reachable states of abstract system can be approximated by computing the least fixpoint of abstract operator \( post_\beta \), which in practise performed by model checking the abstract transition system induced by abstraction function \( \alpha_\beta \).

Definition (Abstract transition system) Given a concrete transition system \( TS = (S, R, I) \), and a Galois connection \((\alpha, \gamma)\) from the concrete domain \((D^S, \subseteq)\) to the abstract domain \((D^\alpha, \subseteq)\), the abstract transition system \( TS^\alpha = (S^\alpha, R^\alpha, I^\alpha) \) induced by the abstraction function \( \alpha \) is defined as follows:

1) \( S^\alpha = \alpha(S) \),
2) \( R^\alpha = \{(s^\alpha, s'^\alpha) \ | \ \exists s, s' \in S(s^\alpha = \alpha(s) \land s'^\alpha = \alpha(s')) \land (s, s') \in R\} \),
3) \( I^\alpha = \{s^\alpha | \exists s \in S(s \in I \land s^\alpha = \alpha(s))\} \).

However, the complexity of constructing the abstract transition system is exponential to the number of predicates being used. Hence it is still expensive for large systems, especially when the number of predicates becomes large.

Cartesian abstraction [3] [4] is on top of predicate abstraction. The basic idea is to ignore the correlations between different predicates (i.e. relations between the elements of different bitvectors). For instance, if we have a set of abstract states \( \{(1, 1, 0), (0, 0, 0)\} \), then by using Cartesian abstraction we will get a trivector \((*,*,0)\) where symbol * means ‘unknown’. However, after Cartesian abstraction, we lose the information that the first element is always the same with the second one.

Given the vector domain \( D^\alpha_1 \times \ldots \times D^\alpha_n \) (in our case each \( D^\alpha_i = \{0, 1\} \)), the abstraction function of Cartesian abstraction is defined by:

\[
\alpha_c : 2^{D^\alpha_1 \times \ldots \times D^\alpha_n} \rightarrow 2^{D^\alpha_1 \times \ldots \times D^\alpha_n},
\]

\( \alpha_c(V) = \Pi_1(V) \times \ldots \times \Pi_n(V) \),

where symbol \( \Pi_i(V) = \{v_i | (v_1, ..., v_n) \in V\} \). Accordingly, \( \Pi_i(V) \) is of one of the three forms: \( \{0\} \), \{1\} and \{0, 1\}. For simplicity, we can write 0 for \( \{0\} \), 1 for \{1\} and * for \{0, 1\}. Then the abstract domain of Cartesian abstraction can be represented as \( D^\alpha = \{0, 1, *\}^n \).

The concretization function is defined by:

\[
\gamma_c : 2^{D^\alpha_1 \times \ldots \times D^\alpha_n} \rightarrow 2^{D^\beta_1 \times \ldots \times D^\beta_n},
\]

\( \gamma_c((W_1, ..., W_n)) = W_1 \times \ldots \times W_n \),

where each \( W_i \) is of one of the three forms: \( \{0\} \), \{1\} and \{0, 1\}. In practice, we will combine predicate abstraction and cartesian abstraction together and get a new abstraction as follows:

\[
post_\beta^{\alpha_c} = \alpha_c \circ post \circ \gamma_\beta \circ \gamma_c
\]

With Cartesian abstraction, we can approximate each predicate independently and it is possible to carry out a source-to-source abstraction. Considering abstraction refinement, there are two sources of imprecision in the abstract model: the spurious traces arise because the set of predicates is not rich enough to distinguish between certain concrete states and the spurious transitions arise because the Cartesian abstraction ignores the correlations between predicates. Spurious traces are eliminated by adding extra predicates, which are obtained by analysing the counterexample produced by model checking the abstract transition system. Most recent techniques for counterexample analysis and abstraction refinement is the Craig interpolation [11] [12]. Spurious transitions are eliminated by adding constraints to the abstract transition relations.

Predicate abstraction has been proved useful for the safety properties of sequential programs if the properties can be
expressed as a quantifier-free boolean combination of the predefined predicates. However, it does not work well for invariants containing quantifiers. Besides, for concurrent programs or systems, predicate abstraction does not scale either. It is also difficult to find a minimal set of predicates which are strong enough to prove or disprove certain invariants.

E. Application: Verifying Speculative Linearizability

In concurrent computing, linearizability is an important correctness property of implementations of concurrent objects. Linearizability reduces the difficult problem of reasoning about concurrent data types to that of reasoning about sequential ones. However, it is extremely difficult to devise efficient and robust implementations of a linearizable object, as well as to reason about them. To cope with this problem, speculative linearizability is proposed as a modular framework for devising, implementing and reasoning about concurrent object implementations in a scalable way [13]. In speculative linearizability framework, a concurrent object can be implemented by several speculation phases, each of which works under a certain situation and if one speculation phase fails, then it will abort and switch to another phase in a safe manner. Speculative linearizability ensures that these abort-switch actions do not break the linearizability constraints and the composition of these speculation phases that are speculatively linearizable is itself speculatively linearizable. In the following, I will define the long and complex definition of speculative linearizability in an intuitive way.

Consider a set of clients accessing an object through invocation procedures, provided by the object whose execution ends by returning a response. The object is accessed sequentially if every response to an invocation immediately follows the invocation. In a sequential execution, the subsequences containing only and all invocations are called histories. Histories are used to explain the responses. However, when the object is accessed concurrently, invocations and responses of different clients may be arbitrarily interleaved and overlapped. The sequences of invocations and responses that may be observed at the interface of an object that is accessed concurrently are called traces. A trace is said to be well-formed if it respects the real accessing of the concurrent object for each client. A well-formed trace \( t \) is linearizable if we can associate, to each response \( r \) returned by a client \( c \), a history \( h \) such that:

1. \( r \) is the response obtained in the sequential execution represented by history \( h \);
2. all invocations in the history \( h \) are invoked in the trace \( t \) before \( r \) is returned;
3. \( h \) ends up with the last invocation of \( c \);
4. for all histories \( h_1 \) and \( h_2 \), either \( h_1 \) is a prefix of \( h_2 \) or \( h_2 \) is a prefix of \( h_1 \).

Intuitively, a trace is linearizable if each invocation appears to take effect at a single point in time after the invocation call and before the corresponding response.

Speculative linearizability is an extension of linearizability to traces augmented with a switch action. Consider a concurrent object implemented by two consecutive speculation phases identified by number \( m \) and \( m+1 \), the traces generated by accessing the object concurrently are formed by three kinds of actions: invocation \( inv(c, o, in) \), response \( res(c, o, in, out) \) and switch \( swi(c, o, in, v) \) from one speculation phase to another, where \( c \) represent the client, \( o \) is the identity number of the speculation phase, and \( in \) represents the input to the object, and \( out \) represents the output of the object, and \( v \) in the switch action is the initial value passed by the aborted speculation phase. Then intuitively a well-formed trace \( t \) is said to be \((m, m+1)\)-speculative linearizable if:

1. for switch action of form \( swi(\_m, \_m, v) \), we can associate a history \( h \) to the switch value \( v \) such that the concatenation trace \( t' \) of subtraces of \( t \) obtained by removing switch actions and the well-formed trace represented by \( h \) is linearizable;
2. for switch action of form \( swi(\_m + 1, \_m, v') \), we can associate a history \( h' \) to value \( v' \) such that \( h' \) is a linearization of trace \( t' \) obtained in item 1.

Moreover, it is proved that the composition of two speculatively linearizable phases is speculatively linearizable (i.e. intra-object composition theorem). Thanks to this theorem, we have a modular framework to devise and implement linearizable concurrent objects by focusing on each dimension of speculation independently and we can still make sure that composition of these optimized linearizable speculation phases is linearizable.

In our context, the problem to be solved is how to automatically verify the design of a speculation phase is speculatively linearizable. Since speculative linearizability is a trace property of a concurrent object, so proving speculative linearizability boils down to generating and proving invariants of the concurrent object. In this study, I will apply BIP to devise, implement and reason about concurrent objects based on speculative linearizability framework.

III. RELATED WORK

To be complete, we mention some other related work on safety property verification.

Assume-guarantee is a compositional reasoning approach that enables verifying each component separately and deducing properties of the global system from local ones [6]. Consider a system composed of two components \( S_1 \) and \( S_2 \). \( P \) is a property to be verified on the parallel composition of \( S_1 \) and \( S_2 \), denoted by \( S_1 \parallel S_2 \). The basic assume-guarantee rule is as follows:

\[
(\langle A \rangle S_1 \langle P \rangle) \quad (\langle TRU E \rangle S_2 \langle A \rangle) \quad \vdash (\langle TRU E \rangle S_1 \parallel S_2 \langle P \rangle)
\]

That is, if under assumption \( A \), components \( S_1 \) satisfies property \( P \) and \( A \) is satisfied by component \( S_2 \), then the system resulting from the parallel composition \( S_1 \parallel S_2 \) satisfies the property \( P \). The main difficulty of assume-guarantee reasoning is to find the decompositions into subsystems. The verification complexity highly relies on the way of decomposition, however, finding a good decomposition is not always feasible. Another problem is how to generate adequate assumptions for a particular decomposition automatically, which in practice is difficult.
**DEDUCTIVE VERIFICATION** provides a proof rule to prove safety properties [16]. To prove that predicate $\Phi$ is an invariant of a system $S$, it is necessary and sufficient to find an auxiliary predicate $\Phi^{aux}$ with the following properties:

1) $\Phi^{aux}$ is stronger than $\Phi$;
2) $\Phi^{aux}$ is preserved by every transition of $S$;
3) $\Phi^{aux}$ is satisfied by every initial state of $S$.

However, it leaves the open questions: how to find the auxiliary predicate and how to prove that the auxiliary predicate is preserved by transition relation and initial states. Indeed, choosing the set of reachable states as the auxiliary predicate reduces the problem to checking validity of the first premise. However, computing the reachable states in most cases is very expensive.

**Partial order reduction** aims at reducing the size of state space to be searched in an explicit model checking algorithm [15]. It exploits the independence of concurrently executed transitions, which result in the same state when executed in different orders. Intuitively, if two transitions $t_1$ and $t_2$ are executed consequently but in any order, the system arrives in the same state, so it is not necessary to explore both the $t_1t_2$ and $t_2t_1$ interleavings. However, detecting independence of transitions is difficult.

**IV. RESEARCH PROPOSAL**

Although verification of safety property of sequential programs has achieved enormous progress in the past decades, for component-based systems the achievement is quite limited mainly because of the state explosion problem resulted from concurrency. In this research, I will propose an approach to the compositional verification of the safety property of component-based systems, in particular the BIP systems. As an application, I will study how to apply BIP framework to devise and implement linearizable concurrent objects and how to verify speculative linearizability automatically.

The central problem of safety property verification is to find an invariant of the system, which is strong enough to prove or disprove the safety property to be checked. Additionally we require that if the property is disproved, our approach should be able to display a trace which violates the property (i.e. a counterexample). In the following, I give two proposals that I would like to investigate in this research.

**A. Proposal 1**

This approach follows the counterexample guided abstraction refinement paradigm, but the analysis will be carried out in a compositional manner by exploiting the modularity feature of BIP systems. In the research, I will answer the following two questions:

1) how to abstract and verify BIP systems modularly;
2) when a spurious counterexample is produced, how to refine the abstraction by reasoning about the counterexample.

An abstraction technique that would be used is the predicate abstraction presented in Section D. For this specific technique, some additional problems that have to be solved are the followings:

1) how to find the minimal set of predicates that are strong enough to prove or disprove the properties;
2) how to deal with properties that contain logical quantifiers;
3) when abstracting each component modularly, how to deal with interaction constraints involving different components.

**B. Proposal 2**

By taking into account the architecture features of BIP systems, I propose another approach which essentially performs an abstract reachability analysis. The safety property verification is achieved by concluding that no bad states, which violate the property are reachable. In details, the problems that have to be solved are the followings:

1) how to guarantee the termination of the abstract reachability analysis by using widening and narrowing techniques necessarily;
2) how to refine an analysis when a spurious counterexample is found.

Another two interesting problems related to optimising abstract reachability analysis are:

1) how to achieve as much synchrony as possible by detecting independent interactions between different components such that the searching length could be reduced;
2) how to accelerate the analysis by using pre-established local invariants of each component.

**REFERENCES**


