Model checking of component connectors

Izadi, M.

2 Context and Backgrounds

In this chapter, we introduce the context and the background of the thesis. In Section 2.1, we briefly review the notions of component based systems and coordination. The problem of formal verification of reactive systems and its solution methods, namely, deductive verification, model checking and their combinations are discussed in Section 2.2. Section 2.3 is an overview of a set of advanced techniques for model checking, used in this thesis to deal with the problem of state explosion. In Section 2.4, we describe the framework of temporal logics in the context of the linear or branching time views and explain briefly the reason for selecting the linear time view as our logical framework in this thesis.

2.1 Component Based Systems and Coordination

The concept of *component based systems*, especially component-based software, is a new philosophy or way of thinking to deal with the complexity in designing large scale computing systems[13, 135]. One of the main goals of this approach is to compose reusable components by some glue code. The model or the way in which these *components* are composed is called a *coordination model*. Thus, a component-based system has two main parts: a set of components and a coordinating subsystem.

What are Components? A *component* is defined as a unit of computation that can be used independently and is subject to composition by a third party [135]. To be used independently, a clear distinction of the component from its environment and the other components is required. Thus, the component communicates with its environment through its *interface*. The interface of a component can be defined as a specification of its access points [135]. Therefore, a component is a black box computing system that communicates with the other components and its environment through its interface by which it offers its provided services.

What is Coordination? A component based system needs connections and interactions among its components. The main goal of *coordination* is to manage the interaction among the components. Thus, coordination can be defined as the study of the dynamic topologies of interactions among components, and the construction of protocols to realize such topologies that ensure well-behavedness [15].

The task of coordination can be realized using different ways of communication, management and control strategies, methods of creation and termination of computational activities, and ways of synchronization of components. A set of definite choices for all of these different aspects is referred as a *coordination model*.

Furthermore, a *coordination language* is the linguistic formalism that is used to specify a coordination model. Thus, a coordination language offers facilities for the specification of the ways of controlling synchronization, communication, creation and termination of computational activities, and other aspects of the coordination model.

There are many languages and models for coordination which provide formal description of the glue code for plugging components together. Coordination models and languages can be classified in different categories using (at least) three different aspects [15]:

- *Dataflow-oriented* versus *Control-oriented* and *Data-oriented* Coordination. The task of coordination can be realized by focusing on data, control or the dataflow in the component based system. In other words, the coordinator can enforce its management strategies over the whole system by focusing on the shared body of data and its evolution or on processing and flow of control. The flow of control can generally be considered not only over the flow of data but also over the control points of components. However, the flow of control can in particular, be considered only over the flow of data. Coordination models that realize their strategies by focusing on data or on the flow of control are called *data-oriented* and *control-oriented* coordination, respectively. In particular, those that coordinate the system only by focusing on the flow of data are called *dataflow-oriented* [15, 120, 12]. For instance, Linda [9, 41, 42] uses a data-oriented coordination model, whereas Manifold [24] is a control-oriented and Reo [13] is a dataflow-oriented coordination language.

- *Exogenous* versus *Endogenous* Coordination. A coordination model can consider the components that are under its coordination as open or black-boxes. A coordination model is called *exogenous* if its view about the components is that they are black-boxes that the coordinator has no access to their internal structure and they should be coordinated from outside. In contrast, an *endogenous* coordination model provides a set of primitives that must be incorporated *within* the components [15]. For instance, Linda [9, 41, 42] is based on an endogenous model, whereas Manifold [24] and Reo [13] are exogenous coordination languages.

- Coordination Mechanisms. Based on the mechanisms of coordination and the mediums through which the coordinator enforces its strategies, coordination models and languages can be classified in the four categories [133, 15]:

  - Coordination through *massage passing*. In this type of coordination, components send messages to each other. The connections between components can be defined (1) as a point-to-point model where each massage has a specific origin and target, or (2) as a publish-and-subscribe model where a component can send a message meant for any component having some kind of a specific service.
  - *Event-based* coordination. In this mechanism, a component, called the producer or event source, can create and fire events, the events are then received by other components, called consumers or event listeners, that listen to this particular kind of events [133].
  - Coordination through *shared data spaces*. In this mechanism of coordination, there is a set of shared data spaces that all components can read from or write values to with specific formats, usually tuples like in Linda [9, 41, 42]. This shared data values contain both data and condition (control) fields. The coordination task is realized by the predefined protocol of reading and writing over the shared data spaces by the components.
  - *Channel-based* coordination. A channel is a one-to-one connection that offers two ends, its source and its sink, to components. A component can write by inserting values to the source-end, and read by removing values from the sink-end

of a channel; the data exchange is locally one way: from a component into a channel or from a channel into a component. The communication is anonymous: the components do not know each other, just the channel-ends they have access to [133, 15]. Reo is an instance of a specification language for the channel based coordination models [13].

There are some more formal and mathematical specification or modeling formalisms for coordination, such as interface automata [10], I/O automata [106], and all of the operational models introduced for Reo [13], namely constraint automata [30], our BAR and ABAR models [71, 73, 72], timed constraint automata [17, 18], probabilistic constraint automata [28], stochastic constraint automata [31], intentional constraint automata [52], Reo automata [36, 37], and port automata [90, 96]. In a more fundamental view, each of these formal models by itself can be used as a formalism for modeling coordination systems. Also, according to a high level and very abstract view, all of these formalisms are fundamentally labeled transition systems with some constraints, guards or classifications on their transitions. But their expressive powers for modeling of coordination systems and also, in practice, their efficiency as the specification tools in the context of verification and model checking are different.

Reo is one of the most recent proposed coordination models. (The first paper on Reo was published in 2002 [25].) Reo is a channel-based exogenous coordination language in which complex coordinators are built out of simpler ones [13]. Reo is a dataflow-oriented channel based exogenous coordination model, in which, the glue code is provided by a network of channels obtained through a series of operations that create channel instances from a set of primitive channels and link them in the network of *nodes* [13]. By Reo specifications one can specify the coordinating model in a compositional and hierarchal way. We describe Reo and constraint automata in chapter 3.

2.2 Formal Verification and Its Methods

The main goal of verification methods is to ascertain that an actual system or program satisfies its requirements. In *formal verification* methods one tries to achieve the aim of verification by describing the system using a mathematical model, expressing the requirements as properties of this model and by showing through rigorous mathematical reasoning that the model of the system indeed has the required properties [50, 110]. Thus, the techniques for formal verification can be considered as comprising of three parts [68]:

- a *framework for modeling systems*, which is typically a description language or some sort of a transition system;
- a *specification language* for describing the properties to be verified;
- a *verification method* to establish whether the description of a system satisfies the specification.

From the view point of mathematical logic, if the desired property to be verified is expressed by a formula in a formal language, say ϕ, the process of reasoning about the correctness of ϕ can be done using *proof theoretic* or *model theoretic* methods or a combination of them. In the literature of formal verification, using proof-based methods is called *deductive verification*, while using the model theoretic approach is called *model checking*. In the following subsections, we briefly describe these methods of formal verification and the ways in which they are combined.

2.2.1 Deductive Verification

In *deductive verification*, the system intended to be proved correct is described by a set of formulas called *axioms* and the correctness property is expressed by a formula in the same language of axioms. The process of verification is to derive a proof of the desired correctness property based on the set of axioms. In other words, if the property to be verified, say ϕ, is expressed in a formal language with a well-established proof theory, and the system is specified by a set of formulas Γ, the deductive verification method is to try to find a proof for $\Gamma \vdash \phi$.

Obviously, if we are interested to do the deductive verification automatically, we need to have an implemented automatic theorem prover for the formal language in which the axioms and the desired property are expressed. Typically, it is supposed that an expert (mathematician or logician) uses the theorem prover and helps it to find the best ways of doing the proof. The facts that only experts can use a theorem prover and fully automated theorem provers are not found are of the main disadvantages of the method of deductive verification. Another problem with theorem proving is that it is not particulary suitable in providing *debugging information*, that is, information on the nature and location of errors [139].

An advantage of deductive verification is that it can be used for reasoning about infinite state systems such as systems that work on unbounded data types (for instance the ordinary integers). As another example, theorem proving techniques work well when process structures evolve, that is, new processes can be added and old processes can be aborted during the executions of the system [139]. In these cases, the task of deductive verification can be automated to a limited extent. However, even if the property to be verified is true, no limit can be placed on the amount of time or memory that may be needed in order to find a proof [50].

2.2.2 Model Checking

If the correctness requirements of a formally modeled computing system are given in a mathematical notion, such as linear temporal logic [110], branching time temporal logic [148] or automata on infinite objects [138], an algorithmic model theoretic process called *model checking* [50] can be used to check if the system respects its correctness requirements. In other words, if the property to be verified, say ϕ, is expressed in a formal language L and the system is modeled by a formal model $\mathcal{M}$ (based on the formal semantics of L) the model checking method is to check whether $\mathcal{M} \models \phi$. In brief, the process of model checking a desired property of a system consists of the following three steps:

- *Modeling* all of the possible states and behaviors of the system using a suitable formalism. Typically some sort of finite transition systems such as finite automata over finite

strings or trees, finite automata over infinite words or trees (such as Büchi automata), finite Petri nets, finite Markov chains and so on are used as the modeling formalism. Some model checkers accept the description of the system in some more high-level description languages such as programming languages, say C or Java, or hardware description languages such as Verilog or VHDL, and automatically convert them into the intended transition systems. Obviously, based on the limitations of time and memory or the interesting aspects of the system, each kind of modeling abstracts away several aspects of a real system.

- *Specification* of the property to be verified in a suitable language. Typically some sort of *temporal logics* or *automata-based expressions* are used for the specification of desired properties. There are several kinds of temporal logic in computer science. We discuss the rules of temporal logic proposed in model checking in Section 2.4.

- Running the *verification* algorithm to check whether the model of a system satisfies its specification explores all possible system states in a brute-force manner. Ideally this algorithm is executed completely automatically. In case of a negative result, the user is often provided with an error trace. This can be used as a *counterexample* for the checked property and can help the designer in tracking down where the error occurred. In this case, analyzing the error trace may require a modification to the system and reapplication of the model checking algorithm.

In comparison with the other approaches to verification, the model checking method enjoys some remarkable advantages: 1- It is fully automatic, and its application requires no user supervision or expertise in mathematical disciplines such as logic and theorem proving. 2- When a design fails to satisfy a desired property, the process of model checking always produces a counterexample that demonstrates a behavior which falsifies the property. This faulty trace provides an insight to understanding the real reason for the failure. 3- In comparison with theorem proving, model checking techniques are highly flexible: a large set of analysis and verification questions of many different types can be answered using a single state space based model.

Model checking has shown to be an efficient and easy to use technique in computer systems verification. However, there is a major drawback in using model checking: the model of real system can be extremely large. In the literature this problem is often referred as the *state explosion problem*. In Section 2.3, we describe some techniques that have been suggested to deal with this problem. Another disadvantages of model checking is that it does not work for infinite-state systems.

2.2.3 Combining Deduction and Model Checking

The main advantage of theorem proving is its ability to reason about infinite state spaces and the main benefit of model checking is its ability to be implemented fully as an automatic tool. They can be composed to yield advanced techniques for verification such as abstraction and compositional reasoning.

As a way to deal with the problem of state explosion, *abstraction* techniques have been proposed in order to integrate theorem proving and model checking [49, 104, 102]. The idea

is to reduce the original system to a smaller model via interactive proof techniques. In a second step, the smaller system is analyzed using automatic model checking tools. Usually, the smaller system is obtained by partitioning the original state space via a structure-preserving function between the two state spaces called *abstracting function*. The theorem prover is employed in order to guarantee the abstraction to be sound, that is if the abstract system satisfies a property, so does the original system [117].

Another way to overcome the state explosion problem is *compositional reasoning* techniques: decomposing the process of reasoning about the behavior of a composed system into partial reasonings about the behaviors of its constituents. For this purpose, the system and the desired property both are decomposed into components and local properties. The correctness of each local property of a component is verified by model checking, while showing that the correctness of local properties entails the correctness of the whole desired property is done by theorem proving [86, 123].

2.3 Advanced Techniques of Model Checking

As mentioned before, state explosion is the main problem in model checking. The state space of the model of a program or hardware system can be very large, even for very simple cases such as a program that performs a simple computation over integers.

During the last two decades, many methods have been suggested to reduce the number of states that must be constructed for answering certain verification or analysis questions. Such enhanced state space methods increase substantially the size of the systems that can be verified, while preserving most of the advantages of the model checking method of verification. Equivalence based compositional model checking [86, 123], partial order reduction by representatives [121], the pre-order reduction techniques [60], abstraction methods [49, 104], using the symmetric structure of the models [56], automata theoretic model checking [145, 146, 102] and its on-the-fly enhancements [59], and symbolic model checking [111, 50] are the most important methods for dealing with the state explosion problem. In the following subsections, we describe four model checking techniques which we will in this thesis.

2.3.1 Automata Theoretic Model Checking

Finite automata can be used to model concurrent and interactive systems. Either the set of states or the alphabet set can then represent the states of the modeled system. One of the main advantages of using automata for model checking is that both the modeled system and the specification are represented in the same way. If $\mathcal{A}$ is an automaton modeling the actual system, let $\mathcal{L}(\mathcal{A})$ be the set of all possible behaviors of the system. The specification of the desired property can also be given as an automaton $\mathcal{S}$, over the same alphabet. Then, $\mathcal{L}(\mathcal{S})$ is the set of all allowed behaviors.

Now, the system modeled by $\mathcal{A}$ satisfies the specification $\mathcal{S}$ when $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$ which is equivalent to $\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$. This means that there is no behavior of $\mathcal{A}$ that is disallowed

by $\mathcal{S}$. If the intersection is not empty, any behavior in it corresponds to a counterexample. Thus, the process of automata theoretic model checking can be summarized by the following steps:

- 1- *Modeling* all of the possible states and behaviors of the system using a finite automaton, say $\mathcal{A}$, over finite or infinite words or trees (such as Büchi automata).
- 2- *Specification* of the property to be verified using an automaton, say $\mathcal{S}$. Then, complementing the automaton $\mathcal{S}$, that is, constructing an automaton $\overline{\mathcal{S}}$ that recognizes the language $\overline{\mathcal{L}(\mathcal{S})}$.
- 3- Constructing the *product automaton* that accepts the intersection of the languages of $\mathcal{A}$ and $\overline{\mathcal{S}}$.
- 4- Checking the emptiness of the language of the product automaton. If this language is empty, we announce that the system satisfies the property. Otherwise, every member of the language of the product automaton is a counterexample that can be considered in the debugging of the system.

Moreover, the automaton $\mathcal{S}$ may be obtained using a translation from some specification language such as linear time temporal logic (LTL). In this case, instead of translating a property ϕ into $\mathcal{S}$ and then complementing $\mathcal{S}$, we can simply translate $\neg\phi$ which immediately provides an automaton for the complement language. In this case, the second step (specification step) of the above method is replaced by the following ones:

- 2-1 *Specification* of the property to be verified using a temporal logic formula ϕ.
- 2-2 *Translation* of the formula $\neg\phi$ into an equivalent automaton called $\overline{\mathcal{S}}$.

Thus, the process of automata theoretic model checking in the context of temporal logic specifications is crucially dependent on the following factors:

1- The existence of a translation method from the selected temporal logic into the selected automata formalism.

2- The decidability and complexity of the algorithm to produce the product automaton that accepts the intersection of the languages of two automata of the selected automata formalism.

3- The decidability and complexity of the algorithm to check the emptiness of the language of an automaton of the selected automata formalism.

Fortunately, for the cases of linear time temporal logic (LTL) and Büchi automata, which we will use in this thesis, all of the above three constructions can be done effectively.

2.3.2 On-the-Fly Model Checking

In on-the-fly model checking the algorithm that checks the correctness of a property is integrated into the algorithm that constructs the state space of the system's model. The construction of the state space is immediately stopped when an error against the property is found. Thus, in general, an on-the-fly method speeds up the model checking process in the cases of incorrect systems while it does not for correct systems. Also, this method may reduce the

number of states in another way, by avoiding the construction of those parts of the state space that are not relevant for the property [139].

On-the-fly verification methods can often be combined with other advanced techniques for model checking. For instance, its combination with automata theoretic technique can be explained as follows. Let $\mathcal{A}$ be the Büchi automaton model of a system and $\bar{\mathcal{S}}$ be another automaton describes the negation of the desired property. In the automata theoretic model checking the emptiness of the intersection of $\mathcal{A}$ and $\bar{\mathcal{S}}$ is checked. If the intersection is not empty, a counterexample is reported. Using an on-the-fly method, instead of constructing the automata for both $\mathcal{A}$ and $\mathcal{S}$ first, we construct only the property automaton $\mathcal{S}$. We then use it to guide the construction of the system automaton $\mathcal{A}$ while computing the intersection. In this way, we may frequently construct only a small portion of the state space before we find a counterexample to the property being checked [50]. On the other hand, sometimes it is also possible that the property automaton $\mathcal{S}$, which is obtained in a translation process from a temporal logic formula, itself can be constructed step-by-step using an on-the-fly translation method [59, 107]. Considering both cases at the same time, we obtain a model checking procedure where the product (intersection) automaton is constructed on-the-fly (by constructing both the system and the property automata on-the-fly), during a depth-first search which checks for emptiness.

2.3.3 Symbolic Model Checking

The model-checking procedure described so far relies on the assumption that the transition system has an explicit representation by the predecessor and successor lists per state. Such an enumerative representation is not adequate for very large transition systems. To counter the state explosion problem, the model-checking procedure can be reformulated in a symbolic way where sets of states and sets of transitions are represented rather than single states and transitions [111, 29]. More generally, we can call this approach to deal with the state explosion problem the method of using *packed state spaces* [139].

There are several possibilities to realize model checking algorithms in a packed state space stetting. The most prominent ones rely on a binary encoding of the states, which permits identifying subsets of the state space and the transition relation with switching functions. To obtain compact representations of switching functions, special data structures have been developed, such as *ordered binary decision diagrams* (OBDD) [111, 50, 29]. In this method, the state set, the transition relation over the set of states and the labeling function which assigns proposition sets onto states, all are encoded by OBDD models. Then the verification algorithms and tools have to be modified to work on the packed state spaces instead of ordinary ones.

The efficiency and usefulness of the symbolic representation of the state space using OBDD models and their suitable verification algorithms are crucially dependent on the selected ordering over the set of boolean variables, which is fixed before constructing the OBDD models. It can be shown that for some switching function, using two different variable orderings produces OBDD structures whose sizes differ exponentially [29]. Since for many switching functions the OBDD sizes for different variable ordering can vary enormously, the efficiency of OBDD-based computations crucially relies on the use of techniques that improve a given variable ordering. However, the problem to find an optimal variable ordering is

known to be computationally NP-hard [34]. Thus, the main problem with the symbolic model checking method is to find the best variable ordering.

2.3.4 Compositional Minimization

Compositional model checking is one of the proposed methods for dealing with the problem of state explosion [50, 51, 123]. In the compositional verification of a system, one seeks to verify properties of the system using the properties of its constituent modules. In general, compositional verification may be exploited more effectively when the model is naturally decomposable [128]. In particular, a model consisting of inherently independent modules is suitable for compositional verification. A special case of compositional verification is the method of *equivalence based compositional reduction*. In this method components of a system are reduced with respect to an *equivalence relation* before building the complete system [60, 50, 79, 82].

In order to be useful in model checking, the selected equivalence relation should satisfy two properties: *preservation* of all properties to be verified and being a *congruence* relation with respect to all operators that are used for composing the models. By a congruence relation we mean that the replacement of a component of a model by an equivalent one should always yield a model which is equivalent to the original one. Thus, two main criteria for selecting an equivalence relation to be used for reducing the models are the set of properties to be verified and the composition operators that are used over the models. If using of an equivalence relation to reduce the size of models produces the smallest ones, we refer to the reduction procedure as *minimization*.

For example, in the context of failure based semantic models of the process description language LOTOS, there are two equivalence relations, called *chaos-free failures divergences (CFFD)* and *non-divergent failure divergences (NDFD)*, which satisfy the preservation property for two fragments of linear temporal logic. NDFD preserves all properties that are expressible in linear time temporal logic without next-time operator (called LTL_{-X}) [86]. Also, CFFD preserves all properties that are expressible in linear temporal logic without the next-time operator but with an extra operator that distinguishes deadlocks from divergences (called LTL^{ω}) [141, 142]. Also, it has been shown that CFFD and NDFD are the weakest equivalence relations that preserve the above mentioned fragments of linear temporal logic. In other words, they both produce the minimized transition systems with respect to the preserved temporal logics. Moreover, it has been shown that in the case of standard labeled transition systems CFFD and NDFD are congruences with respect to all composition operators defined in LOTOS [142].

2.4 The Rules of Temporal Logics

To specify and then verify the desired properties of computer and computing systems, we need a logic for specifying properties of the state-transition models. Classical propositional and predicate logics allow to build up complicated expressions describing properties of states.

For *reactive systems*, correctness depends on the executions of the system, not only on (the state of) the input and output of a computation and on fairness issues. *Temporal logic* is a formalism for treating execution path and fairness aspects. Temporal logic extends propositional or predicate logic by modalities that permit to refer to the infinite behavior of a reactive system. They provide a very intuitive but mathematically precise notation for expressing properties of the relation between the state labels in executions.

The underlying nature of time in temporal logics can be either linear or branching. In the linear view, at each moment in time there is a single successor moment, whereas in the branching view it has a branching, tree-like structure, where time may split into alternative courses. Based on which view is chosen, a system of temporal logic is classified as either a linear time temporal logic in which the semantics of time structure is linear, or a branching time logic corresponding to branching time structures as the semantics of the formulas.

The most popular linear time temporal logic that is used in the field of model checking is the propositional version of a temporal logic called *LTL (Linear Temporal Logic)*. In the linear temporal logic LTL, formulas are composed from the set of atomic propositions using the usual Boolean connectives as well as the temporal connective $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next), and U (until).

On the other hand, the most popular branching time temporal logic in the field is a propositional branching time temporal logic called *CTL (Computational Tree Logic)*. Also, there is a more expressive branching time temporal logic that is called *CTL** [55]. The branching temporal logic CTL* extends LTL by the path quantifiers E (there exists a computation) and A (for all computations). The branching temporal logic CTL is a fragment of CTL* in which every temporal connective is preceded by a path quantifier. CTL* is more expressive than both temporal logics CTL and LTL. The expressiveness of CTL is not comparable to that of LTL. For example, the LTL formula $\Diamond\Box p$ is not expressible in CTL, while the CTL formula $A\Diamond A\Box p$ is not expressible in LTL. There are several other temporal logics such as the branching temporal logic $\forall$CTL that is a fragment of CTL in which only universal path quantification is allowed and linear time or branching time μ-calculus.

The discussion of the relative merits of linear versus branching temporal logics in the context of specification and verification goes back to 1980 [148, 147]. There are several papers arguing against or in favor of linear and branching views (for more see [147]). Our choice in this thesis is the *linear time* approach, not because of such kinds of arguments. The main reason for our choice is because we will study *automata theoretic* and *compositional minimization* methods for which linear time temporal logics are more applicable [147, 148, 146, 143, 139].