The State Explosion Problem

Martin Kot

August 16, 2003

1 Introduction

One from main approaches to checking correctness of a concurrent system are state space methods. They are suitable for automatic analysis and verification of the behaviour of systems. In their basic form, they construct a structure that consist of all states that a system can reach, and all transitions that system can make between those states. This structure is called state space. State spaces can be constructed fully automated. Given the state space of a system, there are many practical practical algorithms for answering some verification and analysis questions. Unlike theorem provers, user don't have to provide invariants or variants. It suffices to formulate an analysis or verification question and start a tool.

State space methods have many advantages and sound like almost ideal behavioural analysis and verification technique. Unfortunately they suffer from one big and fundamental problem - state explosion. Almost any system has huge number of states. Often, the size of a state space of a system tends to grow exponentially in the number of its processes and variables. The base of the exponentiation depends on the number of local states a process has, the number of values a variable may store and the extent to which the local states of components are determined by the local states of other components.

The advantages of state space methods motivated researchers. For answering certain verification and analysis questions, many methods have been suggested that reduce number of required states. The size of systems that can be analysed or verified increased significantly. Unfortunately, advanced state space methods can often answer only certain kinds of analysis or verification questions. For other kinds they lose ability to reduce the number of states.

2 Basic Concepts

State space is the tuple (S, T, Δ, S_I), where

- S is a set of states.
- T is a set of structural transitions.
- $\Delta \subseteq S \times T \times S$ is a set of semantic transitions or edges.
- S_I is a set of initial states. It satisfies $S_I \subseteq S$ and $S_I \neq \emptyset$.

Often we include only those states that system can reach during an execution that starts in an initial state. In the case of Petri nets, states are called markings. Execution of structural transitions causes the system to change its state. Semantic transitions model actual changes of state by the system. They can be called occurrences of structural transitions. A semantic transition is a triple (s, t, s'), where $s \in S$ is a start state, $t \in T$ is structural transition, and $s' \in S$ is end state. In the case of Petri nets we use $M[t\rangle M'$ instead of (M, t, M'). $s_0 - t_1 \rightarrow s_1, s_1 - t_2 \rightarrow s_2, \ldots s_{n-1} - t_n \rightarrow s_n$ is often abbreviated to $s_0 - t_1 \rightarrow s_1 - t_2 \rightarrow \ldots - t_n \rightarrow s_n$. $s_0 - t_1 t_2 \ldots t_n \rightarrow s_n$ claims that there are states $s_1, s_2, \ldots s_n$ such that $s_0 - t_1 \rightarrow s_1 - t_2 \rightarrow \ldots - t_n \rightarrow s_n$. $s \rightarrow s'$ means that there is some $t \in T$ such that $s - t \rightarrow s'$. $s \rightarrow^\star s'$ holds iff there is a sequence $t_1 t_2 \ldots t_n$ of structural transitions such that $s_0 - t_1 t_2 \ldots t_n \rightarrow s'$.

The definition allows more than one initial state. Those are states the system may be in when its execution starts. Petri nets have only one initial marking, typically denoted M_I. A state space with more initial states can be transformed to state space with one new initial state.

A state space is finite iff S and T are finite. The size of a finite state space we usually define as $|S| + |T| + |\Delta|$. The state space is finitely branching iff each state has a finite number of output edges (for every $s \in S$, the set $(s \times T \times S) \cap \Delta$ is finite.

If structural transition are not needed by state space method, then a state space is defined as the triple (S, Δ, S_I), where $\Delta \subseteq S \times S$ and $\emptyset \neq S_I \subseteq S$.

A structural transition t is enabled in a state s, iff there is a state s' such that $(s, t, s') \in \Delta$. In Petri nets it is denoted by $M[t\rangle$ (in process algebras $s - t \rightarrow$). A state is a deadlock if no structural transition is enabled in it.

A structural transition t is deterministic iff $\forall s, s_1, s_2 \in S : (s - t \rightarrow s_1 \wedge s - t \rightarrow s_2 \Rightarrow s_1 = s_2)$. Nondeterministic are those transitions, which are not deterministic.

An execution of a system is a finite or infinite sequence $\langle s_0, t_1, s_1, t_2, \ldots t_n, s_n\rangle$ or $\langle s_0, t_1, s_1, t_2, \ldots\rangle$ such that $s_0 \in S_I$ and $s_0 - t_1 \rightarrow s_1 - t_2 \rightarrow \ldots - t_n \rightarrow s_n$ or $s_0 - t_1 \rightarrow s_1 - t_2 \rightarrow \ldots$. A finite execution is incomplete if it ends in a state with enabled structural transitions. A deadlocking execution is finite and ends in a deadlock state. Complete execution is infinite or deadlocking. By CEx we denote set of all complete executions.

A state s' is reachable from a state s iff $s \rightarrow^\star s'$. The set of Petri net markings reachable from the marking M is denoted by $[M\rangle$. A state or semantic transition is reachable, iff it is reachable from some initial state.

State space models only sequential occurrences of transitions. When we wish to model simultaneous occurrence of transitions, we extend state space to true concurrency models as follows. To the set of semantic transitions are added new transitions labeled by nonempty sets of simultaneously occurring structural transitions. Most of the properties, that are usually verified, are insensitive to the difference between interleaving and truly concurrent models.

Often is useful to use higher level of abstraction and so hide some information. For the purpose of discussing typical abstraction mechanisms, we employ two sets of "observables", Π and Σ, together with a special symbol "τ". The properties of the system may be then referred to only with these observables.

Π is a set of atomic propositions. A proposition φ is a function from S to the set $\{False, True\}$ of truth values.

Σ is a set of observable transitions labels. Sometimes they are also called observable or visible actions. The set $\sum$ is called alphabet.

τ is a special unobservable or invisible action. It is used to label those transitions that the specification should not talk about. They model implementation details and are internal to the system. $\tau \notin \Sigma$ holds

To determine the values of the observables, we define two evaluation function.

$\mathcal{E}_\Pi : S \longmapsto 2^\Pi$ assigns to each $s \in S$ the set $\mathcal{E}_\Pi(s) \subseteq \Pi$ of propositions that hold in s. $\varphi(s) = True$ iff $\varphi \in \mathcal{E}_\Pi(s)$.

$\mathcal{E}_\Sigma : T \longmapsto \Sigma \cup \{\tau\}$ gives new names to structural transitions. It need not to be unique. The structural transitions whose occurrences are unobservable have the name τ.

If only transitions labels are needed but structural transition are not, then T in the definition of state space is replaced by Σ or $\Sigma \cup \{\tau\}$. Then $\mathcal{E}_\Sigma$ is discarded because it becomes identical function. The functions $\mathcal{E}_\Sigma$ and $\mathcal{E}_\Pi$ can be extended to executions:

- $\mathcal{E}_\Pi(\langle s_0, t_1, s_1, t_2, \ldots, t_n, s_n\rangle) = \langle \mathcal{E}_\Pi(s_0), \mathcal{E}_\Pi(s_1), \ldots, \mathcal{E}_\Pi(s_n)\rangle$,
- $\mathcal{E}_\Sigma(\langle s_0, t_1, s_1, t_2, \ldots, t_n, s_n\rangle) = \langle \mathcal{E}_\Sigma(t_1), \mathcal{E}_\Sigma(t_2), \ldots, \mathcal{E}_\Sigma(t_n)\rangle$, and
- $\mathcal{E}_{\Sigma+\Pi}(\langle s_0, t_1, s_1, t_2, \ldots, t_n, s_n\rangle) = \langle \mathcal{E}_\Pi(s_0), \mathcal{E}_\Sigma(t_1), \mathcal{E}_\Pi(s_1), \mathcal{E}_\Sigma(t_2), \ldots, \mathcal{E}_\Sigma(t_n), \mathcal{E}_\Pi(s_n)\rangle$.

If a semantic transition has no observable effect, we call it stuttering. Let $\xi = \langle s_0, t_1, s_1, t_2, \ldots, t_n, s_n \rangle$ be a finite or infinite execution. When Π is used stuttering means that $\mathcal{E}_\Pi(s_{i+1}) = \mathcal{E}_\Pi(s_i)$ for some i, and with Σ that $\mathcal{E}_\Sigma(t_i) = \tau$ for some i. Stuttering is infinite iff $\mathcal{E}_\Pi(s_j) = \mathcal{E}_\Pi(s_i)$ or $\mathcal{E}_\Sigma(t_j) = \tau$ for every $j \geq i$. A property is stuttering-insensitive iff its truth value never changes when finite stuttering is added to or removed from a system.

In formalisms called state-based one can refer to the properties of a system only with the elements of Π. It is the case of most temporal logics. In action-based formalisms is Σ used, but Π is not. Most process algebras are in this category. Using both Π and Σ is often redundant. Action information can be encoded into states and vice versa. So we can use state-based methods for action-based verification tasks and vice versa.

The use of Π and $\mathcal{E}_\Pi$, Σ and $\mathcal{E}_\Sigma$ specifies what we can say about the properties of individual states and transitions. Another important issue are their relations over time.

One possibility is that we look separately at each complete execution of the system. So, execution ξ satisfies a property φ, denoted by $\xi \models \varphi$, iff $\mathcal{E}_\Pi(\xi) \models \varphi$ or $\mathcal{E}_\Sigma(\xi) \models \varphi$ or $\mathcal{E}_{\Pi+\Sigma}(\xi) \models \varphi$. The system has the property φ iff $\xi \models \varphi$ for every $\xi \in CEx$. Properties with validity defined in this way, are called linear-time properties. An example of a linear-time property is n-boundedness of a Petri net or reachability of deadlock. On the other way, Petri-net-liveness is not linear-time property.

The second possibility is to look at execution trees. An execution tree of the state space (S, T, Δ, S_I) with the start state $s_I \in S_I$ is rooted edge-labelled graph (V, E, s_I) such that

- V is the set of finite executions $\langle s_0, t_1, \ldots, t_n, s_n \rangle$ of the system where $s_0 = s_I$
- $E = \{(\langle s_I, \ldots, s \rangle, t, \langle s_I, \ldots, s, t, s' \rangle) | \langle s_I, \ldots, s \rangle \in V \wedge (s, t, s') \in \Delta\}$

Properties with validity defined on $\mathcal{E}$-abstracted execution trees are called branching-time properties. The set of branching-time properties does not cover all properties of interest. But, if no abstraction mechanism is used, then the distinction between linear-time and branching-time properties disappears in certain theoretical sense.

In the world of concurrent systems we distinguish between safety and liveness properties. Liveness consist of a set of requirements of the kind "these things should eventually happen". Linear-time safety properties are those properties of $\mathcal{E}$-abstracted executions that have finite counterexamples. A system has a safety property iff no prefix of some execution matches any one of the counterexamples. An example of safety property is n-boundedness of a Petri net place.

Linear-time liveness properties we define as the properties such that $\mathcal{E}$-abstractions of only complete executions qualify as counterexamples. If complete execution with $\mathcal{E}$-abstraction $\langle P_0, a_1, P_1, a_2, \ldots, a_n, P_n \rangle$ is finite, then also its infinite completion $\langle P_0, a_1, P_1, a_2, \ldots, a_n, P_n, \tau, P_n, \tau, P_n, \ldots \rangle$ should be a counterexample. An example of liveness property is "the program will eventually terminate".

For ensuring liveness is often needed an assumption called fairness. Weak fairness (or justice) towards a structural transition t promises that if t is enabled in every state from some point on, then it will eventually occur. Strong fairness (or compassion) requires that if t is enabled infinitely many times, then it should also occur infinitely many times.

3 Structure of the Analysis or Verification Problem

If we want know if system is working correctly we can use for example following methods.

Verification is intended to check or prove if formal system has a formally stated property. A verification technique is one-sided if it can answer for correct systems and for incorrect systems is not terminating or fails. A verification algorithm, given enough time and memory, always eventually terminate with the answer "yes" or "no".

Analysis gives answers to formal questions about the behaviour of a system. Questions need not be "yes/no" and the answer "yes" is not given priority over "no". An analysis algorithm, given enough computer resources, is guaranteed to eventually terminate with a correct answer.

Validation checks if a system behaves as we want. It is informal because compares behaviour of the system to the expectations of the human.

Error detection is supposed to find errors.

A typical framework for computer-aided analysis or verification has the following four components:

1. A formalism for modeling the system.
2. A formalism for stating analysis questions or properties for verification.
3. A formal meaning for the relation "the system has the properties".
4. An algorithm for checking whether a given system satisfies a given specification.

4 Specification and Query Formalisms

State space tools often can automatically produce various statistics on the state space. These are formulated in the terminology of modeling formalism. The user may ask various queries on the state space and so reduce amount of information in statistics. A typical state space query language is suitable for linear-time safety properties. For other properties are statistic and queries less proper. Also, sometimes they are not "semantic" enough or answer a slightly wrong question. Comprehensive statistics and versatile query languages do not go together well with techniques for alleviating state explosion.

To test software modules is often a test bed used . It sends input to modules and check their output. In test beds are often used fact transitions. That are Petri net transitions which are never expected to be enabled. So, if they occur, something gets wrong. Fact transitions may be used to check any linear-time property whose counterexamples can be expressed as regular language over T or $\Sigma \cup \{\tau\}$. For every such transition there is a finite automaton that accepts exactly the finite sequences of transition occurrences that violate the property. This automaton is connected to the system. Then the arrival of a token to a place that corresponds to an acceptance state of the automaton can be detected with a fact transition.

But, fact transitions and finite test automata can not express all linear-time safety properties. The number of all properties is uncountable. On the other hand, specifications of an object or property in any formalism have finite description. It is a finite string of characters, and there are countably many of them. A larger set of properties is obtained with unbounded test automaton. However, such automata cause serious problems to verification algorithms and tools. If an advanced verification method relies on the assumption that the model has a certain special property, then the addition of the fact transition can destroy this property. Using this method is then impossible.

Livelocks (an infinite executions that produces no useful result) cannot be checked with fact transitions. Absence of livelocks can be specified as follows. We specify a set of structural transitions. Livelock is reported iff the system has an infinite execution that contains only a finite number of occurrences of progress transitions. Instead of progress transitions, we can use progress states.

In the action-based case it is natural to declare all transitions with a visible label as progresss transitions. Livelock is then $\mathcal{E}_\Sigma$-abstracted infinite execution that end up with an infinite sequence of τ-transitions. The error executions without τ-transitions form language. If that language is regular, it can be represented as finite automaton. If we connect such automaton called livelock detection automata to the system, we can efficiently on-the-fly detect a large set of livelock errors.

On-the-fly algorithm for detecting livelocks can be obtained by integrating the detection of non-progress cycles with the construction of the state space.

For error detection we can use Büchi automata. We make the joint state space of state space (S, T, Δ, S_I) and the Büchi automaton $(Q, 2^\Pi, \Delta_B, q_I, F)$ with

- $\{(s_I, q) | s_I \in S_I \wedge (q_I, \mathcal{E}_\Pi(s_I), q) \in \Delta_B\}$ as its initial states, and
- the rule $(s, q) - t \rightarrow (s', q') \stackrel{def}{\iff} (s, t, s') \in \Delta \wedge (q, \mathcal{E}_\Pi(s'), q') \in \Delta_B$ determines the set of transitions.

We usually add a τ transition from the deadlock state to itself because Büchi automaton expects an infinite input string.

The detection of an error now corresponds to acceptance by the Büchi automaton. There is efficient algorithm for detecting Büchi acceptance on the fly.

There are more kinds of acceptance states. For example, an error is declared if

- the automaton ever reaches a reject state,
- the system stops while the automaton is in a deadlock monitor state,
- the system livelocks while the automaton is in a livelock monitor state, or
- the automaton passes infinitely many times through an infinite trace monitor state.

Reject states and livelock monitor states can speed up searching of errors. So, they are useful although, everything that can be specified in the linear temporal logic can be checked with Büchi automata.

As formalism for stating properties for verification, temporal logics are often used. There are temporal operators (such as $\Box$, $\Diamond$, $\mathcal{U}$, $\bigcirc$, etc.) designated to specify properties of complete executions.

In LTL logic, system satisfies a formula built of the temporal operators iff all its complete executions satisfy it. A model checking algorithm inputs a state space and temporal logic formula φ and checks whether φ is a true statement about the state space. A model checking algorithm for LTL has worst-case time consumption linear in the size of the state space, and exponential in the length of the formula. LTL model checking problem is PSPACE-complete. Every LTL formula can be compiled to a Büchi automaton that accepts exactly the executions described by the formula. The validity of any LTL formula φ can be checked by constructing a Büchi automaton for $\neg\varphi$ and checking system with it because Büchi automaton is efficient for verifying that no execution has the property described by the automaton. The size of a Büchi automaton may be exponential compared to the size of the LTL formula.

CTL and CTL* logics extends the logic with two new operators (A, E). They quantify formulas over paths of the state space. The system satisfies a formula, iff all of its initial states satisfy it. Also CTL* model checking is PSPACE-hard (CTL* is an extension of LTL). CTL logic has a fast model checking algorithm and has enough expressive power for many verification tasks. Therefore it is very popular in automatic verification.

A process algebra consist of a language for specifying systems, and theory of the behaviour of the system specified in that language. In the context of process algebras, state spaces are called labelled transition systems and defined as (S, Σ, Δ, S_I) where $\tau \notin \Sigma$ and $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$.

The language of a process algebra practically always contains some operators for parallel composition of processes and for hiding of actions. A parallel composition can be defined as follows. Let $L_1 = (S_1, \Sigma_1, \Delta_1, S_{I1})$ and $L_2 = (S_2, \Sigma_2, \Delta_2, S_{I2})$ be LTS. Their parallel composition $L_1 \parallel L_2$ is the LTS (S, Σ, Δ, S_I) such that

- $S = S_1 \times S_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $((s_1, s_2), a, (s_1', s_2')) \in \Delta$ iff either
  - $(s_1, a, s_1') \in \Delta_1, a \notin \Sigma_2$, and $s_2' = s_2$,
  - $(s_2, a, s_2') \in \Delta_2, a \notin \Sigma_1$, and $s_1' = s_1$ or
  - $(s_1, a, s_1') \in \Delta_1, (s_2, a, s_2') \in \Delta_2$, and $a \neq \tau$.
- $S_I = S_{I1} \times S_{I2}$

The hiding operator converts visible actions into τ. At can be defined as **hide** A **in** (S, Σ, Δ, S_I), where A is some set of actions, and

- $\Sigma' = \Sigma - A$, and
- $\Delta' = \{(s, a, s') | (s, a, s') \in \Delta \wedge a \notin A\} \cup \{(s, \tau, s') | \exists a \in A : (s, a, s') \in \Delta\}$.

A relation $\sim \subseteq S_1 \times S_2$ called a strong bisimulation is defined. Its basic idea is that if the LTSs are in strongly bisimilar states and one of them makes a transition, the other can simulate it with own transition with the same label. Two strongly bisimilar systems thus have "the same behaviour". For finite LTS L it is possible to define minimal LTS L_{min} among the LTSs that are strongly bisimilar with L. There is a very efficient algorithm that constructs L_{min} from any given LTS.

In the second category of behavioural equivalences are so-called abstract process equivalences. They abstract away most information about τ-transitions. Many different equivalences of such kind were developed.

The simplest widely used abstract process equivalence is trace equivalence. A trace is sequence of visible actions obtained from a finite execution by removing all states and all τ-symbols. Two LTSs are trace equivalent (written as $L_1 \simeq L_2$) iff they have the same trace semantics (the set of traces). Trace preorder $\sqsupseteq_{tr}$ is defined by $L_1 \sqsupseteq_{tr} L_2$ iff $Tr(L_1) \subseteq Tr(L_2)$. Let L_φ be an LTS whose finite executions are exactly those which do not violate some stuttering-insensitive linear-time safety property φ over Σ. An LTS L has the property φ (written as $L \models \varphi$) iff $L \sqsupseteq_{tr} L_\varphi$ Trace semantics preserves only stuttering-insensitive linear-time safety properties because is defined on the basis of finite executions. But, it preserves all of them.

Every finite LTS can be interpreted as a finite automaton if all states are declared as accepting.

The problems "does $L_1 \simeq_{tr} L_2$ hold" and "does $L_1 \sqsupseteq_{tr} L_2$ hold" are PSPACE-complete. Checking "$L_1 \sqsupseteq_{tr} L_2$" is PSPACE-complete in the size of L_2. As well the problem of finding some equivalent LTS with the smallest possible number of states is PSPACE-hard.

In process algebras it is common to distinguish between deadlock and livelock. A livelock corresponds to an infinite execution with only finite number of visible transitions. We define a divergence trace as finite sequence made from an $\mathcal{E}_\Sigma$-abstraction of the execution by removing all τ-symbols. The set of all divergence traces of an LTS L is denoted with $Divtr(L)$. A state s of an LTS is stable, iff $\neg(s - \tau \rightarrow)$. s refuses $A \subseteq \Sigma$, iff s is stable and has no outgoing transitions labelled with an element of A. A stable failure of an LTS L is any pair (σ, A) such that L has a stable state s that refuses A, and a finite execution that ends at s and has σ as the corresponding trace. $Sfail(L)$ is the set of the stable failures of L. Infinite trace is an infinite sequence made from an infinite execution that contains an infinite number of occurrences of visible transitions by removal of all τ-symbols. $Inftr(L)$ is the set of the infinite traces of L.

Other proper process semantics is failures-divergences model of CSP. $CSPdivtr(L) = \{\sigma \in \Sigma^\star | \exists \rho \leq \sigma \wedge \rho \in Divtr(L)\}$. $CSPfail(L) = Sfail(L) \cup (CSPdivtr(L) \times 2^\Sigma)$. The CSP-semantics of L is the pair $(SCPfail(L), CSPdivtr(L))$

Alternative semantic model based on some kinds of failures and/or divergence traces is Chaos-Free Failures Divergences (CFFD). It is the triple $(Sfail(L), Divtr(L), Inftr(L))$.

Certainly the most well known abstract branching-time semantic model in process algebras is weak bisimilarity. It is similar to strong bisimilarity but when simulating a transition, the simulating LTS may do any number of transitions (including zero) as long as the resulting sequence of the visible actions is the same in both sides.

References

[LPN98] Antti Valmari, The State Explosion Problem, *Lectures on Petri nets: advances in Petri nets* Springer-Verlang, Berlin-Heidelberg, 1998, 429–473