Introduction to Model Checking

Written Abstract for the Seminar "Recent Advances in Model Checking"

Maximilian Weininger

Organizational information

This abstract is based on Chapter 1 of the Handbook of Model Checking [CHVB18]. Section 1 first introduces and motivates model checking, building on [CHVB18, Chapter 1.1]. Section 2 gives some formal definitions from [CHVB18, Chapter 1.2.1 and 1.2.4] which are required for the presentation. Finally, Section 3 provides and outlook for the presentation, which will mainly categorize the other chapters of the handbook as done in [CHVB18, Chapter 1.3].

1 Introduction

Motivation. Our modern world relies on the usage of computer programs. They are crucial for essentially all areas of our lives, including but not limited to "production, transportation, infrastructure, health care, science, finance, administration, defense, and entertainment" [CHVB18, Page 2]. They perform tasks that, if done wrongly, can cost not only enourmous amounts of money, but even human lives, see also [BK08, Chapter 1]. Thus, it is necessary for these computer programs to be *verified*.

What does verified mean? In this abstract, we understand it as *behaving according to the given specification*. This specification can have various forms, many of which are discussed throughout the Handbook. We provide an illustrative motivational example. When a computer controlled agent (be it an autonomous car of a factory robot) is performing a navigational task (i.e. it is moving around), its specification should contain the following two objectives: firstly, it should be *safe*, which means it should not move into obstacles such as walls or humans; secondly, it should eventually *reach* its destination and not be stuck moving in an endless cycle. We will reuse the example of a robot navigating an environment throughout the abstract.

Model checking. To ensure that the agent satisfies this specification, we can employ the classical and powerful approach of *model checking*. The model checking paradigm is illustrated in Figure 1. The core idea is to construct a formal model of the **system** K and the **specification** ϕ. In our example, the system model should contain all possible behaviours of the computer controlled agent, while the specification should describe the two requirements of being safe and reaching the destination. Then, a model checking **algorithm** is employed to check whether the system satisfies the specification (denoted as $K \models \phi$). The result is either a mathematical proof that the system indeed behaves as specified, or a counterexample demonstrating how the specification can be violated.

In Section 2, we define concrete formalisms for system and specification, namely Kripke structure and formula in linear temporal logic, respectively. Further, we recall the key result that for this combination, model checking algorithms exist. However, there are many other more expressive formalisms that have to be employed in certain settings. This is why we speak of the model checking *paradigm*: it is a way of thinking, a way of approaching manifold problems and finding solutions to them. It is not restricted to Kripke structures and linear temporal logic, but in fact there are various ways of instantiating the model checking paradigm with more powerful formalisms. Such formalisms as well as the required algorithms are the content of my presentation and outlined in Section 3.

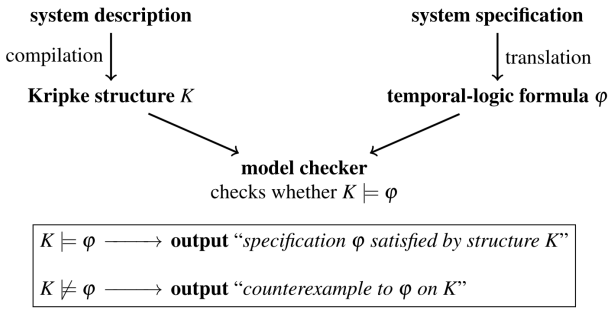

Figure 1: Illustration of the model checking paradigm. Figure taken from [CHVB18, Figure 1].

2 Preliminaries

2.1 System: Kripke structures

Definition 1. *A* Kripke structure *over a set of atomic propositions A is a tuple $K = (S, R, L)$, where S is a finite set of states, $R \subseteq S \times S$ is a set of transitions and $L: S \to 2^A$ is the labeling function.*

Example 1. *We exemplify all parts of the definition on the example from the introduction which is illustrated*[1] *in Figure 2. The state space S contains a description of all possible configurations the system can be in. Concretely, in our example we abstract the environment of the agent as a two-dimensional grid. Then there is a state for every cell of the grid, and hence the state describes the current position of the agent. Formally, we have $S = \{(0,0),\ (0,1),\ (1,0),\ (1,1)\}$. The transition relation R contains all possible ways in which the system can evolve. For example, if the agent currently is in state $s = (0,0)$ (at the origin of the grid), its actions are to move to $p = (1,0)$ or $q = (0,1)$. This is modelled by having $\{(s,p),(s,q)\} \subseteq R$, i.e. by having these two possible state transitions inside the transition relation. Note that in our example all transitions are symmetric, so we also have $\{(p,s),(q,s)\} \subseteq R$. The labeling function L depends on the set of atomic propositions A. These propositions describe relevant properties of system states. In our example, we have $A = \{o,d\}$, i.e. there are two atomic propositions o, indicating an obstacle, and d, indicating the destination. The labeling function tells us that at position $(0,1)$ there is an obstacle and state $(1,1)$ is the destination. Formally, we have $L((0,1)) = \{o\}$ and $L((1,1)) = \{d\}$. Note that for the other states, the labeling is empty, i.e. $L((0,0)) = L((1,0)) = \emptyset$.*

The dynamic behaviour of the system modelled as a Kripke structure is formalized by means of *paths*. A path is a finite or infinite sequence of states, formally $\pi = s_0, s_1, s_2, \ldots$ with $(s_i, s_{i+1}) \in R$ for all $i \geq 0$.

2.2 Specification: Linear temporal logic

Definition 2. *Linear temporal logic (LTL) is a propositional logic over a set of atomic propositions A. It is interpreted over paths. LTL-formulas are defined recursively: every $a \in A$ is a formula in LTL; if ϕ and ψ are LTL-formulas, so are $\neg\phi$, $\phi \wedge \psi$, $\mathbf{F}\phi$ and $\mathbf{G}\phi$.*

A complete definition of LTL is not required for the purpose of this abstract and the presentation. We refer the interested reader to [CHVB18, Chapter 2.3.1]. Here, it suffices to understand what it means to be a logic interpreted over paths and the meaning of the temporal operators **F** and **G**. We will explain these things by means of our running example.

[1] Image built using three license-free pictures. Sources: `https://de.freepik.com/vektoren-premium/netter-roboter-der-hand-cartoon-illustration-winkt_12853813.htm`, `https://de.freepik.com/vektoren-premium/strassensperre-dargestellt_26561729.htm`, `https://de.freepik.com/vektoren-kostenlos/ort_2900811.htm`.

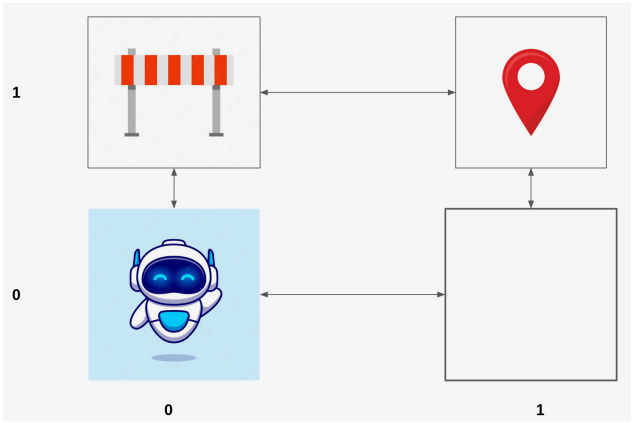

Figure 2: Example of a system modelled as a Kripke structure. See Example 1 for a description.

Example 2. *For the sake of readability, we rename the states of our running example as follows:* $s = (0,0)$, $p = (1,0)$, $q = (0,1)$, $r = (1,1)$. *A path in our running example is a sequence of grid positions. For example, if the robot is always alternating between state s and p, this corresponds to the infinite path* $\pi_1 = (s,p)^\omega = s,p,s,p,\ldots$. *To illustrate all relevant concepts, we define two further paths:* $\pi_2 = s,p,r,(p,r)^\omega$ *goes to the destination while avoiding the obstacle and then alternates between states p and r.* $\pi_3 = s,q,r,(p,r)^\omega$ *is very similar, only differing in that it goes to the destination over the state q which contains the obstacle.*

Intuitively, we want to formalize the fact that π_2 *is the path we want, because it reaches the destination while avoiding the obstacle.* π_1 *is suboptimal because it never reaches the destination; and* π_3 *because it hits the obstacle. To formalize this, we will specify an LTL-formula that is satisfied by* π_2*, but not by* π_1 *or* π_3.

The first part of the specification requires us to reach the destination. This can be encoded by the LTL-formula $\mathbf{F}d$*, where* $d \in A$ *is the atomic proposition labeling the destination and* $\mathbf{F}$ *is the so-called* future *temporal operator. A path* π *satisfies an LTL-formula* $\mathbf{F}\phi$*, if at some point in the path* ϕ *is satisfied. Since* π_2 *contains r and since* $L(r) = \{d\}$*, we have that* $\pi_2 \models \mathbf{F}d$*. Similarly,* $\pi_3 \models \mathbf{F}d$*, but* $\pi_1 \not\models \mathbf{F}d$.

The second part of the specification prescribes that we have to avoid the obstacle. The LTL-formula $\mathbf{G}\neg o$ *formalizes this, where* $\mathbf{G}$ *is the so-called* globally *temporal operator. A path* π *satisfies an LTL-formula* $\mathbf{G}\phi$*, if at all positions in the path* ϕ *is satisfied. Hence,* $\mathbf{G}\neg o$ *is satisfied by a path* π *if and only if for all states x in the path* π *we have* $o \notin L(x)$*. Thus, it holds that* $\pi_2 \models \mathbf{G}\neg o$ *as well as* $\pi_1 \models \mathbf{G}\neg o$*, but* $\pi_3 \not\models \mathbf{G}\neg o$*, because state q is contained in* π_3 *and* $L(q) = \{o\}$.

In summary, our specification can be encoded as the LTL-formula $(\mathbf{F}d) \wedge (\mathbf{G}\neg o)$.

2.3 Model checking algorithms

Given a system in the form of a Kripke structure and a specification formalized as an LTL-formula, we can check whether the system satisfies the specification. Formally, this amounts to verifying that all paths in the Kripke structure satisfy the LTL-formula. There exist efficient algorithms for that, which utilize the fact that a path violating the LTL-formula must have a specific structure and exploiting the relationship of LTL and automata over infinite words [CHVB18, Theorem 2].

Note that in our running example, not all paths satisfy the specification. This means that there are ways in which the system can violate our expectation and either hit the obstacle or avoid the destination. This tells us that the robot is "wrong" and we need to do something about it. One possibility is to synthesize a controller for the robot that only selects certain transitions in every state, namely those that are safe with respect to the property. For example, we can remove all transitions towards the obstacle $(0,1)$ as well as the transition back to the starting state $((1,0),(0,0))$. The resulting Kripke structure satisfies the objective, since

no path in it can reach the obstacle and every path has to reach the destination. Now, the robot is verified to behave as expected.

3 Outlook

We have just seen the basic model checking paradigm: verifying that a system satisfies a specification by applying a sophisticated algorithm. This basic setting is elaborated in Chapters 2-5. There are two main directions of research that extend this, focusing either on the *modeling challenge* or the *algorithmic challenge*.

The modeling challenge. The real world is complex and we need the formalisms we use to describe it to be expressive enough so that they capture all relevant properties. Thus, we can extend the systems to include

- an unbounded number of parallel processes (Chapter 21).
- non-finite state concepts such as nonces and keys in security protocols (Chapter 22).
- multiple actors with conflicting goals. This also lays the basis for controller synthesis (Chapter 27).
- uncertainties, e.g. for modeling randomness required in privacy protocols or abstracting probabilistic events such as sensor faults or biological processes (Chapter 28).
- continuous state-variables, such as time (Chapter 29) or, in the case of cyber-physical systems, temperature or velocity (Chapter 30).

The algorithmic challenge. At the core of model checking are efficient algorithms, be it for the basic problems or the ones with more expressive models. In particular, we have to deal with the state-space explosion problem [BK08, Chapter 2.3] (also known as curse of dimensionality). The presentation will explain this in more detail, but the key issue is that the systems we want to model check are very, *very* large, and hence the basic idea of smartly exploring the needed parts of the state-space (Chapter 5) is often infeasible because of the required memory.

There are algorithmic methods for mitigating this problem, which can roughly be classified as follows:

- Structural methods try to reduce the size of the state-space we have to consider by using (i) partial-order reduction, essentially grouping equivalent parts together; or (ii) compositional reasoning, which allows to reason about the smaller components of the system and lift the resulting insights to the whole system (Chapters 12, 17, 18).
- Symbolic methods represent the system in a more concise way than by remembering every state and transition explicitly. By exploiting properties of the used data structures, the required memory and model checking time decreases drastically. Symbolic methods can use Binary Decision Diagrams (Chapters 7 and 8) or rely on SAT or SMT encodings (Chapters 9, 10, 11).
- Abstraction (Chapters 13-15) tries to reduce a system to a smaller, homomorphic image so that the key properties of the original system are retained in the abstraction.

In the presentation, I will provide more details on the challenges and the ways in which the chapters of the handbook address them. For deciding which chapters you want to present, I also recommend reading [CHVB18, Chapter 1.3] as well as abstracts of the chapters you are most interested in.

References

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.