L&PS – Logic and Philosophy of Science
Vol. IX, No. 1, 2011, pp. 569-577

Software Verification and Scientific Methodology: Models, Regularities, Abstractions

Nicola Angius
Dipartimento di Storia, Scienze dell'Uomo e della Formazione
Università degli Studi di Sassari
e-mail: nangius@uniss.it

ABSTRACT. Essential traits of *model checking*, a prominent formal method utilized in computer science to predict future behaviours of software systems, are examined here in the framework of the *model-based* paradigm of scientific reasoning. Models that model checking techniques enable one to develop are shown to satisfy logical requirements expressed by the set-theoretic view of scientific models. It is highlighted how model checking algorithms are able to isolate *law-like generalizations* holding in the model under given ceteris paribus conditions and concerning software executions. Furthermore, abstraction methodologies utilized in model checking to decrease the state space of complex models are taken to be instantiations of the general process known as *Aristotelian abstraction* characterizing empirical modelling. Finally, the methodological interest of the model-checking techniques is emphasized in connection with the debate concerning the epistemological status of computer science.

1. Software verification in computer science

Software verification methodologies developed in theoretical computer science are formal tools by means of which software code is statically examined

Published by L&PS – Logic and Philosophy of Science
http://www2.units.it/ ~episteme — ISSN: 1826-1043

in order to determine whether it satisfies specific requirements. *Model checking* (Baier and Katoen 2008) is a model-based formal method enabling to advance hypotheses concerning future behaviours of software and hardware systems. Preliminary, one builds a *system specification*, in the form of a transition system and representing some target computational system capable of interacting with the environment (called a reactive system). A *property specification*, articulating a property that one would like the target system to fulfil, is subsequently formalized with an appropriate temporal logic, either a linear time logic (*LTL*) or a computation tree logic (*CTL*). Finally, the model checker algorithm checks whether the temporal logic formula holds in the model of the system. Essential traits of model checking are here examined in the context of the general philosophy of science views about models, law-like statements and abstractions of empirical systems. It is maintained that system specifications are akin to *empirical models* that act as surrogates of the target software system with the aim of testing a set of hypotheses concerning the program's behaviours. This claim is supported on the basis of an examination of the structure and uses of the computational models used in *model checking* in relation with the structure and uses of empirical models in scientific practice. Furthermore, property specifications positively verified by means of model checking algorithms are acknowledged as *regularities* which, *according to the formal model*, predict and explain future behaviours of the target system.

Runtimes of the verification algorithm are determined by the number of states of the transition system, the state space being too large may constitute a limitation for a state-space search. This problem, known as *state explosion problem*, suggests that simplified models be used to perform the algorithmic verification when highly structured software systems[1] are involved. *Data abstraction*, the practice of removing negligible information in a transition system in order to reduce its state space, is here examined as an *abstraction* of *Aristotelian* type by means of which one is able to build *minimal models* of complex software systems.

Accordingly, the conjectural knowledge of computer programs and hardware systems that model checking enables one to attain involves the deployment of methodologies which, in their essential features, are on a par with traditional methodologies one deploys in physics and other natural sciences for predictive and explanatory purposes.

[1] That is, most non-trivial systems.

Let us initially focus on system specifications given, in model checking, by a Kripke structure and examine under which respects a Kripke structure can be understood as a scientific model.

2. Model checking and scientific method

A Kripke structure (from now on KS) is a labelled transition system given by the four-tuple $M = (S, S_0, R, L)$ where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation which must be total, that is, $\forall s \in S, \exists s' \in S$ such that $R(s, s')$ and $L: S \rightarrow 2^{AP}$ is a function that labels each state in S with a set of atomic propositions (subsets of AP) that are true in that state. A path of a KS is an infinite sequence of states $\pi = s_0, s_1, s_2...$ such that $s_0 \in S_0$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$ (Clarke et al 1999, p. 14). Fig. 1 shows a simple KS M for a traffic light controller.[2] In this example, $S = \{s_0, s_1, s_2\}$; initial state s_0 is the unique element of S_0 pointed by an incoming arrow; transition relations are depicted by arrows from state to state and are given by $R(s_0, s_1)$, $R(s_1, s_2)$, and $R(s_2, s_1)$. Atomic propositions set AP is given by the set $AP = \{Green, Yellow, Red\}$ and each state in S of the KS is labelled by a single element of this set (abbreviated in the picture by "**G**", "**Y**", and "**R**").

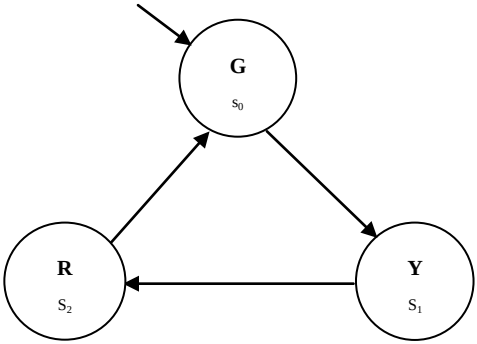

Fig. 1**:** *KS M for a traffic light controller.*

Let us now turn to examine the nature of models developed in model checking in the framework of the ontological status of scientific models in general. The

[2] The example of the traffic light controller used throughout this paper is taken from Clarke et al (1999), Ch. 13.

word "model" refers, in scientific practice, to entities of different nature, such as physical objects, sets of descriptions, sets of equations, or a combination of them. A conception of models advanced by Patrick Suppes (1960), who aims to account for all uses of the word "model" in scientific practice, identifies models with special set-theoretic structures. According to Suppes, the meaning of the word "model" is unique in all fields of science, while the differences that exist between areas such as physics, social sciences or econometrics, concern the way they are used in scientific research. Suppes acquires the notion of set-theoretic model from mathematical logic, as a non-empty set of objects and a non-empty set of relations and operations between objects. Suppes exemplifies this notion of scientific model by reference to a classical particle mechanics model $F = (P, T, s, m, f)$, where P is a set of particles, T is an interval of real numbers corresponding to elapsed times, s is a function from the Cartesian product $P \times T$ of particles in P and time intervals T to positions of particles, m is a mass function defined on the set of particles, and f is a force function defined again on $P \times T$.

A Kripke structure M for a reactive system is a model in the sense understood by Suppes since, as we noted above, it is a set-theoretic structure $M = (S, S_0, R, L)$, which includes a set of objects (states) S (with designated subset S_0), a relation R between these objects describing trajectories in a state-space, and a functional relation L on states identifying properties of those objects. Accordingly, computational models one builds in model checking conform to the set-theoretic analysis of scientific models advanced by Suppes.[3]

Suppose now one is interested in assuring that the traffic light would eventually stop vehicles in a crossroad; this simple property may be formalized in *CTL* as ***AF*** *Red*, which means that, for every paths (**A**) starting at some initial state, there will finally (***F***) be a state where *Red* holds. The *model checking algorithm* determines the set of states in S that satisfy the property specification f: $\{s \in S \mid M, s \vDash f\}$. In the present example the algorithm will end by stating that $M \vDash \mathbf{AF}$ *Red*, that is, by saying that structure M satisfies the formula. Universally quantified temporal logic formulas (obtained using path quantifier ***A***) that have been positively checked by the model checking procedure express regularities concerning the model executions (and the target system behaviours, provided that the model is an accurate representation of the involved system) (see Angius and Tamburrini 2011). This stems from the fact that the

[3] Angius and Tamburrini (2011) contains a comprehensive analysis about the ontology of models in model checking, the semantic relations of such models with the represented software systems, and the surrogative reasoning provided by KSs and enabling to draw law-like hypotheses.

algorithm checks whether the involved property holds in all computational paths starting at some initial state. For example, the statement ***AF*** *Red*, holding in the traffic light controller KS, asserts that *for every computational path* in the model there will be a state wherein *Red* holds true and therefore that the traffic light *will always* eventually turn red.

KSs (and consequently their target systems) are often required to satisfy specific *ceteris paribus* conditions to make law-like statements of this kind hold *for every computational path*. In model checking, *fairness constraints* force KS's behaviours such that specified unfair behaviours be not travelled during computation (Baier and Katoen 2008, pp. 129-139). Unfair behaviours usually express a malfunctioning due to a hardware fault. In the traffic light controller of Fig. 1 a fairness constraint may require, for instance, that a never ending self-loop on state 1 will not ever occur. A deadlock of this kind might be caused by a damage in the internal circuit of the traffic light. However, the model checking of the traffic light controller is involved in the verification of the software rather than in the verification of the final implemented system. Failures caused by hardware errors are of no interest at this stage of verification.

A fair KS is a 5-tuple $M = (S, S_0, R, L, F)$ where $F \subseteq 2^S$ is a set of fairness constraints, given by a set of states which must be visited in a fair path (Clarke et al 1999, p. 33). Suppose now a piece of software actually satisfies the required property, the model checking algorithm might yield a negative response if unfair paths are allowed. Fairness constraints formalize the assumption that the involved computational system works properly and that no errors, due to hardware implementation, occur. Accordingly, in order to check whether a property specification expresses a regularity concerning all reactive system's behaviours one has to assume *ceteris paribus* conditions of stability pertaining the involved computational system.

A further difficulty dealing with the formal verification of program specifications comes from the state explosion problem. Except for very simple programs, KSs are too complex to be handled by the model checking algorithm with available computational resources. In most cases, abstract models are used instead. In the context of complex empirical system modelling, a *minimal model* is a model wherein all details not useful to study the phenomenon of interest are not included (Batterman 2001). Abstract structures used in model checking, on the other hand, *encompass only data that are necessary to check the interested temporal formulas* with available computational resources. Phenomena to be studied are here executions of the target software system. Let us see.

In *data abstraction* one specifies a subset *A* of the system's data set *D* by establishing a function *h* mapping from actual data values to a small set of abstract data values. This function is determined *with respect to those variables that are involved in the temporal specification one is involved to check*. The KS *M* for a traffic light controller illustrated in figure 1 has one single variable, *colour*, which ranges over domain *D* = {*red*, *yellow*, *green*}. Set *AP* is given by equations *colour* = *green*, *colour* = *red* and *colour* = *yellow* with which states of *M* are labelled (using in the picture abbreviations **G**, **Y,** and **R**). Suppose now one aims at simplifying structure *M* to check property ***AF*** *Red* and defines an abstract domain *A* = {*stop*, *go*} and a function *h* such that:

$$h_{colour}(d) = \begin{cases} stop & \text{if } d = red \\ stop & \text{if } d = yellow \\ go & \text{if } d = green \end{cases}$$

Abstract domain *A* determines an abstract set of atomic proposition *AP*° = {*colour*° = *stop*; *colour*° = *go*}; *AP*° induces the KS *M** depicted in figure 2 below.

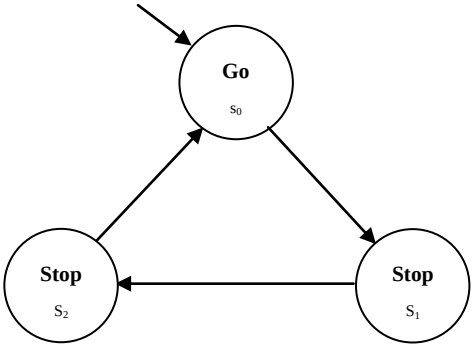

Fig. 2**:** *KS M* for a traffic light controller*

One can, at this point, merge together those states that have the same label, namely those states labelled with equation *colour*° = *stop* , thus obtaining the abstract structure *M*° illustrated in figure 3 (Clarke et al. 1999, pp. 195-199).

The minimal model approach, also known as *Aristotelian abstraction* (Frigg and Hartmann 2006), is defined by Nancy Cartwright (1994) as an op-

eration by means of which one "strip[s] away – in one's imagination – all that is irrelevant to the concerns of the moment in order to focus on a single property or set of properties as if they were separate" (p.187). A typical Aristotelian abstraction is given by the classical mechanics model of the planetary system, wherein planets are taken to be objects only having shape and mass, regardless of their other properties. In model checking, abstract structure $M°$ represents a traffic light controller as having only the function of letting vehicles and pedestrians either go or stop. These functions are considered as separate in the model with the aim of focusing on property ***AF*** *Red*.

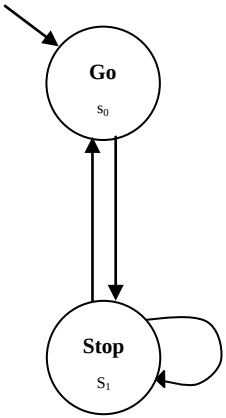

Fig. 3**:** *An abstract KS M° for a traffic light controller*

Let us conclude this note on model checking by drawing some conclusions upon the "Philosophy of Computer Science" and concerning the epistemological status of the discipline.

3. Towards a scientific paradigm in computer science

The Philosophy of Computer Science (**PCS**) is characterized by ontological and epistemological reflections concerning computational systems and methodologies involved in the examination of such systems (Turner and Eden 2008). The practice of building abstract empirical models representing reactive software systems and of drawing regularities concerning the program's behaviours has been here highlighted as a trait of epistemological interest in model checking. Predictions and explanations concerning computational sys-

tems are accomplished according to a set-theoretic model, exemplifying, in the computer science domain, the so-called *model-based approach* which typifies scientific reasoning dealing with complex empirical systems (Magnani, Nersessian and Thagard 1999).

Positively verified regularities are, in model checking, model based hypotheses that still need to be tested by observing actual software executions. In *software testing*, empirical hypotheses of this kind can be falsified by letting the target system run from some initial state and for a specified interval of time Δt, observing executions during Δt, and comparing resulting outputs with expected outputs given by the property specification (Ammann and Offutt 2008).

An additional purpose of **PCS** is shedding light upon the debate about the epistemological status of computer science. Eden (2007) distinguishes three epistemological paradigms in **PCS**: a mathematical paradigm understands computer science as a branch of mathematics, that is, as a deductive, *a priori* knowledge about programs that proceeds by means of some proof; a "technocratic paradigm", promulgated by software engineers, takes computer science to be an *a posteriori* and probabilistic inquiry about programs; and a scientific paradigm which places computer science on a par with empirical sciences, pursuing both deductive and inductive reasoning about programs. Introducing formal models for software systems, advancing hypotheses concerning the program's future behaviours that are proved to hold in the model, and subsequently testing those hypotheses by observing actual program executions place the study of computational systems under a scientific paradigm according to which both deductive and inductive reasoning are involved.

RIFERIMENTI BIBLIOGRAFICI

Ammann, P. and Offutt, J. (2008): *Introduction to Software Testing*, Cambridge: Cambridge University Press.

Angius, N. and Tamburrini, G. (2011): "Scientific Theories of Computational Systems in Model Checking", *Minds and Machines*, 21, pp. 323-336.

Baier, C. and Katoen, J. P. (2008): *Principles of Model Checking*, Cambridge, MA: The MIT Press.

Batterman, R. W. (2002),: "Asymptotics and the Role of Minimal Models", *British Journal for the Philosophy of Science*, 53, pp. 21-38.

Cartwright, N. (1994): *Nature's Capacities and Their Measurement*, Oxford: Oxford University Press.

Clarke, E. M., Grumberg, O. and Peled D. A. (1999): *Model Checking*, Cambridge, MA: The MIT Press.

Eden, H. A. (2007): "Three Paradigms of Computer Science", *Minds & Machines*, 17, pp. 135-167.

Frigg, R. and Hartman, S. (2006): "Models in science", *Stanford Encyclopedia of Philosophy*, available at http://plato.stanford.edu/entries/models-science/.

Magnani, L. , Nersessian, N. and Thagard, P. (1999): *Model Based Reasoning in Scientific Discovery*, Dordrecht: Kluwer.

Suppes, P. (1960): "A Comparison of the Meaning and Uses of Models in Mathematics and the Empirical Sciences", *Synthèse*, 12, pp. 287-301.

Turner, R. and Eden, A. (2008), "The Philosophy of Computer Science", *Stanford Encyclopedia of Philosophy*, available at http://plato.stanford.edu/entries/computer-science/