Model-based Abductive Reasoning in Automated Software Testing
(Former Draft)

Nicola Angius
Dipartimento di Storia, Scienze delle’Uomo e della Formazione
Università degli Studi di Sassari, Italy
nangius@uniss.it

Abstract
Automated software testing using model checking is in this paper epistemologically analysed in order to argue in favour of the model-based abductive reasoning paradigm in the computer science field. Preliminary, it is shown how both deductive and inductive reasoning are insufficient to determine whether a given piece of software is correct with respect to specified behavioural properties. In Automated software testing computational models are algorithmically checked to find paths that satisfy or falsify the properties of interest; such paths are used to select executions to be observed in software testing. Models developed in model checking are acknowledged as scientific models according to the set-theoretic approach to models, and isomorphisms between such models and the target system are shown to hold. A study case exemplifies how, using different models built upon the same system data set, contradictory results can be obtained in software testing. This is used to show the hypothetical nature of analogical models in automated software testing. The model assumption – algorithmic check – software testing course is understood as the abduction – deduction – induction circle defining the selective abduction and turned to isolate a set of model-based hypotheses about the target system behaviours. A manipulative abduction process is finally recognized in the practice of adapting, abstracting, and refining models that do not provide successful predictions.

1. Introduction into the Problem of Correctness
Establishing at which extent programs meet the task requirements they were encoded for is one of the leading problem in theoretical computer science research. The problem of correctness (Boyer and Moore 1981) is, indeed, the problem of determining whether a software system satisfies a given set of specifications which express, in a proper language, certain software requirements. In the software verification field, the correctness of software code can be assessed either statically, by an a-priori examination of instructions that does not involve the running of the program (static code analysis), or dynamically (dynamic code analysis), by launching the program and observing its actual executions. Both static and dynamic analysis are affected by essential limitations (as illustrated in the next section) which call for the extensive use of models representing and simplifying the computational system to be examined. Aim of this paper is understanding the epistemological role played by such models in the software verification process.

Current trends in computer science research and in software engineering applications see static and dynamic code analysis used together to get through the drawbacks of the latters (see for instance Godefroid et al 2008, Artho and Biere 2005). Model checking automated software testing (Callahan, Schneider, and Easterbrook 1996) combines software testing (Ammann and Offutt 2008)
the common and throughout used dynamic code analysis method, with model checking (Bayer and Katoen 2008), a prominent and model-based static analysis technique. Automating software testing using model checking is seen, in the present study, as an application, in the computer science domain, of the model-based reasoning (Magnani et al. 1999) which characterizes most of scientific research. Angius and Tamburrini (2011) advance the thesis that computational models in the software verification framework play similar epistemological roles of scientific models. In the context of the discussion about the epistemological status of computer science (Eden 2007), Angius (2012) hints at the idea of a model-based reasoning paradigm for the discipline. This idea is here spelled out in the light of the distinguishing features characterizing model-based reasoning since the pioneering works in (Magnani et al. 1999).

Model-based reasoning, as being a form of abductive reasoning, is considered divergent to deductive and inductive reasoning (Nersessian 1999). Next section shows how both deductive and inductive reasoning in computer science, exemplified respectively by theorem proving and software testing, are insufficient to provide suitable solutions to the problem of correctness. The thesis maintained throughout the paper is that computational models used to overcome the inner limitations of the deductive and inductive approaches allows for an abductive reasoning of software systems. Abductive movements in scientific models are granted by established isomorphic relations between parts of the model and objects of the represented system (Nersessian 1999, Van Fraassen 1980). The third section introduces Kripke structures as scientific models used in model checking to represent target software systems and it displays isomorphisms between states of such structures and programs data. The model checking algorithm is introduced and it is shown, by means of a study case, how different results about a program’s correctness are obtained in case different models are used. The hypothetical character of such models highlights, in section four, how a selective abduction (Magnani 1999) is involved in automated software testing, turned to formulate a set of hypotheses, to perform a selection among them, and to test them empirically. Finally, the manipulative character of abduction (Magnani 2004) is underlined in the process of finding a suitable model which predictions are coherent with observed executions of the target program. Manipulations of this sort involve re-orderings of computational paths of the model, abstractions, and opportune refinement of those abstractions.

2. Reasoning about programs and the need of models
Providing a mathematical proof of program correctness is the aim of Hoare logic and its developments (Van Leeuwen 1990, Ch. 15). In Hoare’s original paper (Hoare 1969), a static code analysis is conceived by providing a method of axiomatization of programs together with a set of rules defining a program execution1. According to this approach, a program is correct with respect to a given specification, if the specification, conveniently expressed in the language of the formal system, is derivable from the axioms using the inference rules. Hoare proved the partial correctness

---

1 Axioms and rules are of the form \{P\}C\{Q\} where \(C\) is a program statement and \(P\) and \(Q\) are propositions about the program’s data: given the precondition \(P\), instruction \(C\) will yield postcondition \(Q\). To give a couple of examples, the assignment axiom schema \(\vdash \{P^x=v\} x:=v \{P\}\) states that if the proposition \(P^x=v\) (where all the free occurrences of \(x\) are substituted by \(v\)) hold then, given the assignment \(x:=v\) to variable \(x\), also the original proposition \(P\) holds (Hoare 1969, 577). As a rule of inference, consider the rule of composition: \(\{P_1\}C\{P_2\}\); \(\{P_2\}C^*\{P_3\}\) \(\vdash \{P_1\}C;C^*\{P_3\}\). The rule states that given two commands, the first bringing from precondition \(P_1\) to postcondition \(P_2\), and the command \(C^*\) bringing the program from \(P_2\) to \(P_3\), the composition of instructions \(C;C^*\) will bring the program from the state expressed by \(P_1\) to the state expressed by \(P_3\) (Hoare 1969, 578).
of his axiomatic system, since the termination of computations is not comprised; Hoare logic developments during the 70’s and the 80’s included total correctness. However, proving total correctness calls for the introduction of elementary arithmetic into the formal system’s language, and this makes incompleteness of the system inevitable (Van Leeuwen 1990, 848). In Hoare’s paper, computer science is presented as an “exact science” able to provide a “deductive reasoning” on computer programs. Theorem proving of programs’ specifications is today a fundamental research area of theoretical computer science; nonetheless, incompleteness results constitute, from an epistemological viewpoint, a significant difficulty to support the thesis that deductive reasoning is able to provide a suitable knowledge of software systems. Furthermore, the complexity of the resulting axiomatic system representing non-trivial programs demands for the use computer-aided proofs. The use of automated theorem proving is involved in the epistemological debate questioning the a-priori nature of proofs carried out by a computer (see for instance Arkoudas and Bringsjord 2007).

Software testing is the main dynamic code analysis method used by software developers (Amman and Offutt 2008, 3); it involves the actual execution of the program to be tested. Given a (set of) specification(s) the basic idea is to launch the given program and to observe the resulting executions in order to see whether they cope with the specification. A terminating execution is recorded as an input-output couple and it is called a test case; test cases are collected into a test suite. A test suite is said to satisfy a specification when all input-outputs couples in the test suite are a subset of the input-outputs pairs the specification express correct program behaviours with. A test case not comprised within the specification pairs constitutes a failure; a failure is an incorrect execution violating the specification of interest. The observation of a failure is indicative of presence of a fault in the program’s code. The fundamental drawback of software testing is that observing a finite number of executions may detect the presence of a fault in case a failure is recognized but it cannot assure for the absence of faults if no failure is executed. This problem, known as the Dijkstra’s dictum (Dijkstra 1970), can be understood, from an epistemological point of view, as a problem of enumerative induction. Given a program P and a specification S expressed in a universal form (of the form “for every execution of program P, property K holds), any finite number n of executions can ever satisfy S. Execution n + 1 can always contain a failure. Inductive reasoning on software systems as well seems to be insufficient to provide a satisfactory knowledge on computational artefacts. This will be clarified by the following example.

Consider a simple program class called FileADT; classes are parts of a program which are encoded to accomplish a specific task. A class is composed of a set of methods, each performing a subpart of the desired task. The FileADT class is aimed to add lines of text into a given file; it is composed of the four following methods (replaced by comments):

\[
\begin{align*}
open (tFile) & \text{ - Opens the text file called tFile} \\
\text{close (tFile) } & \text{ - Closes tFile} \\
\text{assign (tString) } & \text{ - Assigns a new value to tString} \\
\text{write (tString) } & \text{ - Writes the line of text stored in tString into tFile}
\end{align*}
\]

2 The termination of commands C is not guaranteed by the while rule which may result in a never-ending self-loop. This may be avoided by introducing a variant into the while rule, that is, a variable which numerical value decreases at each reiteration of the loop until it reaches a predefined value. When the variant equals such value, the loop terminates (Van Leeuwen 1990, 845).

3 This study case is an adaptation of an example provided in (Ammann and Offutt 2008, 76).
The `open` method opens the file named `tFile`, the `assign` method assigns a value to the `tString` variable, i.e. a line of text which may be input from the user, the `write` method adds into the opened file the line of text stored in the `tString` variable, and the `close` method closes the file, in. The order with which the `FileADT` class calls its methods is essential to achieve the desired outputs: the `write` method should be called, each time, after the `assign` method, otherwise the program will either halt yielding an error message (if the `tString` variable is empty) or it will keep on adding the same line of text (if the `tString` variable is assigned already). The present program is meant to call methods in the order `[open()] – [assign()] – [write()] – [write()] – [close()]`. Consequently, a specification $S$ for the `FileADT` class might be “every `write()` is executed after an `assign()`”. An execution of the kind `[open()] – [write()] – [assign()] – [close()]` constitutes a failure of the `FileADT` class with respect to $S$. The induction problem in software testing is that by observing, any finite times, the execution sequence `[open()] – [assign()] – [write()] – [write()] – [close()]`, it cannot be concluded that “for every execution of the system, all `write()` methods are executed after an `assign()`”. At any next execution, the sequence `[open()] – [write()] – [assign()] – [close()]` can be observed.

The problem of induction in software testing comes from the fact that, as in the context of natural sciences, it is not feasible to launch the examined program using all potential inputs, since they are potentially infinite (think about an embedded system taking inputs from the environment) (Ammann and Offutt 2008, 14). This rises the difficulty of choosing which inputs to execute the system with. This problem, known as the controllability problem, (Ammann and Offutt 2008, 14), is the problem of specifying certain coverage criteria indicating which computational paths to cover, that is, to execute and observe in order to evaluate the system with respect to the specification(s) of interest (Ammann and Offutt 2008, 17). Since observing software executions allows only to detect the presence of faults but not their absence, coverage criteria are a means by which to find potential failures to be observed among the executed computations. Finding counterexamples to the involved specifications calls for the use of models. Such models can assume very different natures, depending on the formal technique chosen to generate coverage criteria with and going from directed graphs to sets of logical expressions.

Next section focuses on the use of transition systems to represent programs and on model checking to automatize the process of selection of executions to be observed. Before that, a first main conclusion can be drawn. Both deductive and inductive methodologies in software verification turn out to be, per se, insufficient to reason about computer programs. The complexity of the target system and the need of guessing potential behaviours of a such system are at the basis of the usage of representing models both in the computer science domain and in the model-based reasoning approach in natural sciences (Nersessian 1999).

3. Models in Model Checking Automated Software Testing
Observing a program’s executions enables to detect only the presence of failures; for this reason coverage criteria are meant as selecting those executions containing a violation of the interested specification. Test cases are, in this way, asked to satisfy given test requirements (Ammann and Offutt 2008, 17). In order to test the `FileADT` class against specification $S$, a test requirement $T$ would require to observe all those executions that are violations of $S$, that is, all those runs of the

---

4 Other undesired behaviours of the class can be excluded by different specifications, such as “a `close()` is not called after a `close()` unless and `open()` is executed after the first `close()`”, or “a `write()` is not called after a `close()` unless an `open()` is called after `close()`” (Ammann and Offutt 2008, 76).
system such that no assign method is called before a write method. When dealing with articulated specifications and complex systems involving an enormous number of potential executions, specifying the executions to be observed is not an easy task. Model checking (Baier and Katoen 2008) automatizes this process by means of a deep-first search algorithm operating on a model of the system. Model checking is a static software verification method developed independently; the use of model checking to automatize software testing is becoming a standard and powerful approach (Callahan, Schneider, and Easterbrook 1996; Hong, Cha, Lee, Sokolski, and Ural 2003).

The main idea of the technique can be sketched as follows. The model checking algorithm receives as inputs a transition system representing the target software system and temporal logic formula formalizing the specification of interest. The procedure yields as output a yes or no answer, stating respectively whether the formula holds true or does not hold true in the model, together with a set of either witnesses or counterexamples. In case of a positive answer, the set of witnesses are the set of paths of the transition system satisfying the temporal formula; counterexamples are paths in the model showing a violation of the formula and are exhibited in case of a negative answer of the procedure. Witnesses and counterexamples represents executions that, respectively, satisfy and violate the specification and that thus, are going to be observed in the software testing phase (Hong, Cha, Lee, Sokolski, and Ural 2003, 232-233). The whole process is now analysed in details under the light of the model-based reasoning approach paradigm.

Software systems are represented in model checking as transitions systems of different kind, such as Kripke structures, Büchi automata (Clarke et al. 1999) and transition system with action names (Baier and Katoen 2008, chapter 2). A Kripke structure (KS) is a labelled transition system, that is, a transition system provided with a function that labels each state with propositions that are true in that state. Typically, labelled propositions are equations assigning values to the system variables characterizing each state. A KS \( K = (S, S_0, T, L) \) is defined by a set \( S \) of states and a set \( S_0 \neq \emptyset \) of initial states, together with a total transition relation \( T \subseteq S \times S \) between states\(^5\), and the function \( L: S \rightarrow 2^{TP} \) labelling states with subsets of the propositions set \( TP \) which are true in that state. A path \( \pi \) of a KS is the infinite sequence \( \pi = s_1, s_2, s_3, \ldots \) of states such that \( s_0 \in S \) and \( T(s_n, s_{n+1}) \) holds for all \( n \geq 0 \). Fig. 1 shows a KS \( M \) for the FileADT class. Here circles are states of the KS and arrows are transitions between states; state \( s_I \) is the only initial state; states are labelled with methods called in each state.

Instead of applying logical rules on program statements, in order to reason about properties of the FileADT class, a model about such system is built and experiments are performed on the said representing model. According to Nersessian (1999), in order to be able to perform simulative reasoning, a model is meant to be a “structural analog of a real word or imaginary situation”, that is, it “embodies a representation of the spatial and temporal relations among […] the entities depicted” (Nersessian 1999, 11). The Kripke structure \( M \) depicted in Fig.1 can be properly understood as a structural analogue of the FileADT class informally reported above. The class is a piece of a program composed of three methods; states of \( M \), on the other hand, stand for each of the methods of the class. The KS contains a little bit more of information about the class: it indicates (all potential) orders with which methods can be called during computation. \( M \) embodies a temporal representation of the depicted entities, that is, of the states standing for methods in the class.

---

\(^5\) The transition relation \( T(s_n, s_{n+1}) \) between two states \( s_n \) and its successor \( s_{n+1} \) is required to be total to ensure that for every \( s_n \) there exist a state \( s_{n+2} \) such that \( \{s_n, s_{n+2}\} \in T \). Kripke structures are able to model, in this way, never ending computations.
Coherently with the mode-based reasoning paradigm, it is properly the act of representing all possible ordering of method’s calling that allows to check for correct behaviours among potential behaviours of the software system. This opportunity is a characteristics of mental models (Johnson-Laird 1983) only.

![A KS M for FileADT class.](image)

Mental models, as opposed to material models, can assume very different natures; they can be propositional (mathematical, linguistic) or non-propositional (graphic, schematic, fictional), or even hybrid (like labelled graphs) (Nersessian 1999, 12). Angius and Tamburrini (2011) provided an epistemological analysis of the nature and the usage of Kripke structures in model-checking; KSs were there understood as scientific models according to the set-theoretic approach on scientific models. Following Suppes (1960), a scientific model of this sort is a non-propositional set-theoretical entity defined by a set of objects together with a set of relations among those objects and turned to define the object of the modelled system. A set-theoretic model is a structure \( m = \langle A, R_i, f_j, a_k \rangle \) where \( A \) is a non empty set, \( R_i, i \in I \), is a family of relations, \( f_j, j \in J \), is a family of functions and \( a_k, k \in K \), is a family of elements of \( A \) (French and Ladyan 1999). A KS \( K = (S, S_i, T, L) \) defines a target software system by introducing a set of objects \( S \) (states), a set of relations (transitions) \( T \) between those objects, and a function \( L \) which describes the objects by means of propositions which are true for singular objects.\(^6\) Function \( L \) determines the hybrid nature of models in model-checking.

Analogue models enable to perform experiments on the representing model rather than on the represented empirical system; the surrogative reasoning (Swoyer 1991) capability of analogue models is related to the opportunity of establishing isomorphic relations between elements of models and their target systems (Nersessian 1999, 15). In the context of the set-theoretic approach on scientific models, Van Fraassen (1980) defined the empirical adequacy of a model to the

\[ A KS K = (S, S_i, T, L) \] does not differ that much from the set-theoretic model \( F = (P, T, s, m, f) \) for the classical particle mechanics Suppes (1960) refers to. Model \( F \) specifies a set of particles \( P \) and a set of time intervals \( T \) together with the space function \( s \), the mass function \( m \) and the force function \( f \) which all define objects in \( P \) (Suppes 1960, 6).
represented system as a property such that substructures of the model are isomorphic to the *appearances* of the observed phenomena (van Fraassen 1980, 64). Van Fraassen calls appearances whatever results from the observation of phenomena from the observer’s point of view; appearances are what “appear” to the observer and hence they are an already interpreted version of data. A structural isomorphism is, in Redhead’s words (Redhead 1980), “a situation in which related objects have related properties”; more precisely, two structures $m = \langle A, R_i, f_j, a_k \rangle$ and $m' = \langle A', R'_i, f'_j, a'_k \rangle$ of the kind defined above are said to be isomorphic iff there is a bijective function $g$ from $A$ to $A'$ such that for all $x$ and $y$ in $A$ and all $R_i$, $R_i xy$ iff $R_i g(x)g(y)$. Van Fraasen’s (1980) examination of the function of representation in terms of an isomorphic relation between a formal model and a set of observable phenomena can be assimilated into the process through which a KS is built. A KS is extracted from a set of logical formulas that determines the values of the system’s variables. The observable phenomena of a reactive system are represented by assuming a specific set of system variables and by assigning (previously measured) values to those variables. The representational relation of a KS is relative, as assumed by Van Fraassen for a scientific model in general, to an initial representation of the empirical system, wherein measurement reports are formalized in a certain way. Consider Fig. 2 below, which illustrates how an extraction of a KS forms a set of logical formulas: elements of the KS, also known as states, correspond to the logical formula that assigns values to the variables that characterize that state; if a transition relation occurs between two states, a “transition” also occurs between the corresponding logical formulas in the form of ordered pairs of variables valuations (or also in the form of a Boolean conjunction of the two logical formulas).

- $s_1 \iff x = \text{open}$
- $s_2 \iff x' = \text{write}$
- $R(s_1, s_2) \iff x = \text{open} \land x' = \text{write}$

*Fig. 2* An example of extraction of a KS from a set of equations.

Specifications to be checked are formalized in model-checking by means of *temporal logic* formulas (Kroger and Merz 2008) which are able to express truth values of logic formulas as the given transition system evolves. Linear time logics (LTL) formulas assume that each state of the system possess a single successor state; computation tree logics (CTL), which take current states to be followed by different successor states (following thus a branching time structure), is best suited for formalizing temporal properties of the KS in Fig. 1. A CTL formula is given by a first-order logic formula obtained by propositions in TP and Boolean connectives, temporal operators, and path quantifiers. Temporal operators specify in which state, during the evolution of the system a given property hold. Common temporal operators include $X$ (next), $F$ (finally), $G$ (globally), $U$ (until), and $R$ (release). The two path quantifiers $A$ and $E$ are used to quantify the involved logic formula respectively over all the paths or over at least one path. For instance, given a KS $K$ and general first-order logic formula $\rho$, $K \models AF\rho$ means that for every paths starting at all initial states of $K$, there will finally be a state wherein $\rho$ holds. And the CTL formula $EG\rho$ is satisfied by $K$ in case there exists at least one computational path in $K$, starting at an initial state, such that all the
states of the path satisfy \( p \). Atomic logic formulas like \( p \) are satisfied by a given state \( s \) of a KS, that is, \( s \models p \), when \( L(s) = p \). For instance, in KS M above, \( s_1 \models \text{open} \) but \( s_1 \not\models \text{close} \).

Specification \( S \) for the FileADT class (“every write() is executed after an assign()”) can be formalized by the CTL formula \( \text{AG}(\text{write} \rightarrow \text{assign}) \). This formula means that for every state \( (G) \) of every path starting at an initial state \( (A) \), if the next state is labelled with \( \text{write} \), then present state is labelled with \( \text{assign} \). A CTL model checking algorithm is launched to compute whether structure M in Fig.1 satisfies \( \text{AG}(\text{write} \rightarrow \text{assign}) \). The algorithm will end by stating that M \( \not\models \text{AG}(\text{write} \rightarrow \text{assign}) \) and counterexample \( s_1, s_2, s_4 \) will be showed. Path \( s_1, s_2, s_4 \) constitutes the failure the FileADT program will be made to execute. In case the program performs such execution, it is concluded that the program contains the respective fault; otherwise the conclusion is that the built KS is not an adequate representation of the system, the model is modified accordingly to be further checked against subsequent specifications. Fig.2 provides an alternative KS N representing the FileADT program and wherein the showed counterexample has been removed. Clearly, \( N \models \text{AG}(\text{write} \rightarrow \text{assign}) \).

---

7 The semantics of first order formulas \( K \models \varphi \) is inductively defined, as usual, over the structure of \( \varphi \). That is, \( K \models \neg \varphi \) iff \( K \not\models \varphi \), \( K \models \varphi \land \psi \) iff \( K \models \varphi \) and \( K \models \psi \), and so on.

8 The CTL model checking procedure proposed by Clarke et al. (1999) is a labeling algorithm that, given a KS K and temporal formula \( \varphi \), labels each state of K with subformulas of \( \varphi \) that are true in such state. The algorithm proceeds step by step starting from atomic formulas in \( \varphi \) and then going to molecular formulas until the whole formula is processed (Clarke et al 1999, P. 7). To show a simple case, to check whether \( K \models EX g \), the algorithm will first label states in K where \( g \) holds, then it will label states which successor state was previously labeled by \( g \). \( K \models EX g \) if there is path wherein at least one state is labeled by \( EX g \).
Finding counterexamples of specifications of interest is, in most common software, quite an hard task. Model checking automatizes this process by an algorithmic check on a representing model of the involved program. This proficiency of this form of simulative reasoning is related to the isomorphic relations between paths in the graph and executions of the program together with an *inner dynamics* (Nersessian 1999, 12, 19) possessed by Kripke structures. Inner dynamics concern the way such mapped paths are related to each other obtaining more complex paths. Structure M and structure N above models the same basic paths in different ways. This shows how Kripke structures involve “tacit assumptions” (Nersessian 1999, 15, 19) about the potential executions of the represented software system. Such assumptions nonetheless enable to draw new conclusions about properties of target system.

Kripke structures are interpretations of their target software systems insofar as they hypothesize potential behaviours of such systems. A given program can be modelled in different ways and, depending on the model used to perform the model checking algorithm, different results concerning the specifications of interests are gained. As shown before, N ⊨ AG(Xwrite → assign) whereas M ⊭ AG(Xwrite → assign). Furthermore, notice how structure N does not satisfy other specifications of interests previously mentioned, such as “a close() is not called after a close() unless and open() is executed after the first close()”, or “a write() is not called after a close() unless an open() is called after a close()”. It can be seen how this path-properties are fulfilled by structure M in Fig. 1 instead. Evaluating a set of specifications with respect to a given software system by hypothesizing, within a computational model, all potential behaviours of such system, underlines the abductive role played by kripke structure in the whole process.

4. Model-Based Reasoning About Programs

For a given software system and a set of specifications, Automated Software Testing using Model checking has been shown to be the practice of formalizing specifications using suitable temporal logics, introducing a model in the form of a KS representing the target system, performing an algorithmic check of the temporal logic formulas within the built model, and finally launching the program looking for witnesses or counterexamples generated by the model checking in case the model respectively fulfils or does not fulfil the temporal formula under consideration. Specifications of the form of S for the FileADT example, and quantifying over all possible paths, can be understood as hypotheses concerning behavioural properties of the examined computational system. Aim of the software verification process is to test such set of hypotheses. Software testing performs this task by looking for counterexamples of such hypotheses, that is, it tests specifications by trying to falsifying them: specifications are thus scientific, that is, falsifiable hypotheses (Northover et al. 2008). In order to see whether a specification can be falsified, a KS is built that hypothesize, in its turn, all potential behaviours of the represented software system. A KS is often called a system specification indeed.

Given a set of specifications, the task performed by Kripke structures is selecting which hypotheses hold true in the model and which do not. The abductive role of a KS relies in its being an hypothetical structure properly conceived to obtain explanations of observable phenomena. Once a given temporal logic formula quantifying over all possible paths (as specification S above) has been positively verified with the built KS, it assumes the epistemological status of a model-based regularity able to predict future executions of the target software system (Angius and Tamburrini 2011, ).
Magnani (1999) introduced the notion of *model-based abduction*, as distinguished from sentential abduction which is based on the propositional knowledge of an agent. In model-based abduction the selective and explicative role is carried out by a model as conceived in the previous section. According to Magnani’s extensive examination of the nature and uses of abduction in scientific practice, abduction can be *creative* or *selective* depending on whether it results in the creation of new hypotheses or in the selection of a subset of pre-constituted hypotheses.

The selective character of the model-based abduction involved in automated software testing results in “the making of a preliminary guess that introduces a set of preliminary hypotheses, followed by deduction to explore their consequences, and by induction to test them with available patient data (1) to increase the likelihood of a hypothesis noting evidence explained by that one, rather than by competing hypotheses, or (2) to refute all but one” (Magnani 1999, 222). Model checking automated software has been examined as the practice of introducing a “preliminary guess”, that is, an interpretation of the software system to be examined in the form of a KS, which “introduces a set of preliminary hypotheses”, i.e. temporal logic formulas formalizing properties to be fulfilled by the target system and that are interpreted over the specified KS. By launching the model checking algorithm, all dynamic consequences of the KS are deductively explored to see whether the formalized formulas hold in the KS. By exploiting witnesses and counterexamples yielded by the algorithm, obtained results are inductively tested by software testing, that is, by observing the actual software executions. Aim of the whole process is finding evidences (observed executions) that falsify a given specification or, in case the corresponding temporal formulas has been positively verified in the KS, executions that are coherent with the specification.

Coherently with the model-based reasoning paradigm, the model checking automated software testing methodology involves an abduction – deduction – induction process. The abductive nature of the model construction process allows new conclusions about future execution of the target system to be obtained. By hypothesizing all potential executions of the examined program, it is feasible to find witnesses and counterexamples to the interested temporal formulas. This highlights the ampliative character of the knowledge achievable with KSs. The ampliative knowledge provided by models in model checking is responsible for the non-monotonicity of the model-based abduction (Magnani 1999, 222): by adding or changing paths in the model, different conclusions are obtained, as shown before for structures M and N. Also, from hypothetical knowledge only *uncertain* conclusions can be drawn. As exemplified above, in case observed executions do not cope with results obtained with model checking, the involved KS is modified accordingly (Carson 2002).

Modifications and manipulations of models take constantly place in the model checking whole process. Magnani (2004) puts *manipulative abduction* alongside with selective and creative abduction in scientific research. Manipulative abduction deals with the practice of opportunely modifying the representing model so as to obtain hypotheses that are actual explanations of observed phenomena. The need of manipulation arises when abducted hypotheses turn out to be not coherent with observed new phenomena or just in case a direct abduction process is not practicable because of the complexity of the phenomenon under study. In such cases, significant observed phenomena, such as a regularity or an unexpected event, are used to modify the involved model, by introducing, modifying, or re-ordering the allowed behaviours. Models involved in this process play the role of *epistemic mediators* (Magnani 2004, 233-238) which goals concern, among others, simplifying the target system and/or coping with the lack of system data by modelling hypothetical
system behaviours. The *abduction – deduction – induction* process is thus modified as the circle *manipulation – abduction – deduction – induction – manipulation – abduction – deduction – induction* – etc… In this way different model manipulations leads to different observations or to the observation of the target system under different perspectives, and new unexpected observations leads to further manipulations of the model.9

The need of manipulating Kripke Structures arises, in the first place, in case output yielded by the model checking algorithm are not coherent with observed executions in the software testing phase. This is the case exemplified above for the *FileADT* program, model M and specification S. The algorithm terminated stating that $M \not\models \textit{AG}(\textit{Xwrite} \rightarrow \textit{assign})$ and exhibiting a counterexample. It was assumed that any corresponding failure was detected among observed actual executions and structure M was modified accordingly. The new model N was able to fulfil the interested temporal logic formula and observed phenomena became coherent with the model assumptions. An unexpected event, i.e. the absence of the expected failure, brought to modify the representing model by re-ordering all paths representing potential computations of the target program.

A further reason to revise a KS comes from complexity issues. The model checking algorithm, although being decidable for models involving only discrete variables, might not terminate in sufficient time and space resources if launched on a model involving a large number of states and transitions.10 In such cases, the complex model is simplified by applying specific techniques that reduce the state space of the KS. Angius (2012) provides an extensive epistemological analysis of the state space reduction techniques and he argues that they can be understood in terms of common abstraction and idealization methodologies used in scientific research to abstract and idealize empirical models in scientific practice. In *data abstraction* (Kesten and Pnueli 2000) a subset of data involved in the specification to be tested is selected from the system data set. A new KS is built from the data subset: since states are defined over the system’s data, the new model will involve a smaller number of sets.11 The model is so manipulated to re-order transitions between the new subset of states.

Abstract KSs are usually manipulated several times before it becomes feasible to successfully perform the model checking algorithm. Given a KS P and an abstract KS $P^\circ$ and a temporal logic formula $f$, it is required that all the computational paths of P be simulated by paths in $P^\circ$ to ensure that if $P^\circ \models f$ then also $P \models f$. This however will not guarantee that all paths in $P^\circ$ simulate a path in P; $P^\circ$ is called an *over approximation* in case it contains paths not comprised within P. *Abstraction refinement* (Wang et al. 2006) is a methodology turned to re-order paths of over approximated abstract KSs to eliminate non simulating paths. This process usually results in the construction of a hierarchy of abstract simulating structure, from an over approximation to the original un-abstracted KS, among those it is chosen one that does not contain relevant non-simulating paths and such that it is processable with available computational resources.12 Such a model is, for this reason, called a *deciding approximation*. In the context of the model-based

---

9 Magnani (2004) calls to these processes *perceptual and inceptual rehearsal* (pp. 238 – 239)
10 Common model checking algorithms are linear wrt the temporal formula’s length and the set of states $S$. For instance, the requirement of the algorithm generally used for CTL formulas is, for a *CTL* formula $f$ and a KS $K = (S, S_0, T, L)$, time $O(|f| \cdot |S| + |R|)$ (Clarke et al. 1999, 38).
11 Angius (2012) analyses data abstraction under the light of the Aristotelian abstraction of scientific models, according to which a model encompass the minimum of information necessary to study a phenomenon of interest.
12 Abstraction Refinement is compared in Angius (2012) with the multiple model idealization approach used in population biology to build hierarchies of models each pursuing a different epistemological desideratum.
reasoning framework, an over approximation is an *epistemic mediator* aimed at simplifying the software system to be examined.

In conclusion, the abductive processes involved in model checking automated software testing are practical only if KSs are opportunely manipulated. Each manipulation is followed by a new algorithmic search and by new tests, and executions to be detected are ultimately dependent on the improving model modifications. In case unexpected executions are observed, the manipulative abduction process starts again by an ulterior modification of the computational model until a deciding approximated and adequate model is obtained.

5. Conclusions
The analysis of software systems of non-trivial complexity involves the construction and exploitation of computational models (Turner 2009). Such models are opportunely built, modified, and simplified in order to obtain adequate predictions about future computations of the represented system. This process characterizes all the approaches to the problem of correctness. In theorem proving, an axiomatic system formalizing all potential behaviours of a software system is a model of such system. Provable theorems of the axiomatic system represents computations allowed by the software system only if the model is an adequate representation of the target system. This fact needs to be assessed by testing the software system. Theorem proving so goes along with software testing. Model checking cannot be used alone as well. Positively verified temporal formulas hold in the model, they still need to be tested empirically.

This paper analysed model checking automated software testing; however, the epistemological considerations drawn so far can be extended to the computer science general domain. Due to the complexity of programs, deductive and inductive reasoning alone are not sufficient to evaluate their correctness. In both cases, representational models are required. Such models embrace more information than available system data, insofar as the same computations can be modelled in different ways. The implicit additional information comprised within such models, allows for the abductive movements described in the paper, that is, the opportunity of isolating a set of hypotheses concerning the represented system’s behaviours, be them proved theorems in Hoare Logic or temporal logic formulas in model checking. Such model-based hypotheses are subsequently tested using the software testing techniques. If the predictions provided by the said hypotheses are coherent with observed executions, it is concluded for the adequacy of the computational model with the target program. Otherwise, the model is manipulated in such a way that deducible consequences are satisfactory explanations of observed phenomena.

The main thesis maintained throughout this paper is that reasoning about the correctness of programs is a model-based abductive and manipulative reasoning. This conclusion introduces a novelty in the standard classification of the epistemological paradigms of computer science as a mathematical discipline, an empirical discipline, or a scientific discipline (Eden 2007). Some (Gruner 2011, Angius 2012) argue that computer science and software engineering fit the scientific paradigm in their being a combination of deductive and inductive reasoning. This paper, by endorsing the scientific paradigm for computer science, highlights the *creative process* involved in the abductive – deductive – inductive circle. The (descriptive and prescriptive) methodological conclusion of this study is that given a complex software system about which it is not feasible to

---

13 In asynchronous concurrent programs, for instance, computations are carried out simultaneously but they are not clock-synchronized so that they can be represented with alternative orderings in separate models.
provide a faithful representation, building models which introduce assumptions about their target systems allows to acquire new knowledge about those systems. Such model-interpretations of the examined software systems are first evaluated by drawing certain model-based hypotheses that are then inductively tested. In case the hypothetical model is able to provide suitable explications of observed phenomena, it can further be used to obtain whatsoever prediction about computations of interested of the represented program.

References


