Discovering Empirical Theories of Modular Software Systems. An Algebraic Approach.

NICOLA ANGIUS*
PETROS STEFANEAS°
*Dipartimento di Storia, Scienze dell'Uomo e della Formazione
University of Sassari, Italy
nangius@uniss.it
°Department of Mathematics,
School of Applied Mathematical and Physical Sciences
National Technical University of Athens, Greece
petros@math.ntua.gr

Accepted Manuscript in Muller, V. (ed.) (2016), *Computing and Philosophy: Selected Papers from IACAP 2014. (Synthese Library)*, pp. 99-115. Berlin: Springer.

Abstract
This paper is concerned with the construction of theories of software systems yielding adequate predictions of their target systems' computations. It is first argued that mathematical theories of programs are not able to provide predictions that are consistent with observed executions. Empirical theories of software systems are here introduced semantically, in terms of a hierarchy of computational models that are supplied by formal methods and testing techniques in computer science. Both deductive top-down and inductive bottom-up approaches in the discovery of semantic software theories are refused to argue in favour of the abductive process of hypothesising and refining models at each level in the hierarchy, until they become satisfactorily predictive. Empirical theories of computational systems are required to be modular, as modular are most software verification and testing activities. We argue that logic relations must be thereby defined among models representing different modules in a semantic theory of a modular software system. We exclude that scientific structuralism is able to define module relations needed in software modular theories. The algebraic Theory of Institutions is finally introduced to specify the logic structure of modular semantic theories of computational systems.

Keywords: Philosophy of Computer Science; Semantic View of Theories, Modelling, Scientific Structuralism, Abstract Model Theory.

1. Introduction: The Need of Empirical Theories of Software Systems

Aim of the software evaluation processes is *predicting* the future behaviours of the examined systems in order to know whether those behaviours are consistent with the required software specifications. Evaluations of programs' correctness might involve the mathematical approaches provided by formal methods (Fisher 2011), the more empirical practices of software testing (Ammann and Offutt 2008), or a combination of both. In any case, one would like to get to a set of sentences yielding predictions of the system's future computations. This paper is concerned with the construction of *theories* of software systems that be adequately predictive with respect to their target systems' behaviours. We focus on the process of *discovery* of those theories, that is, on the development of theories from the availability of computational models that have been verified or tested during some software evaluation phase. *Abstract model theory* is finally utilized to define the structure of predictive theories of modular systems.

Programs are abstract, human-made, entities, about which it is in principle possible to acquire an a-priori knowledge. Formal methods have been developed in theoretical computer science with the aim of performing a static code analysis, not involving the execution of the implemented software system. Benefits of formal methods concern the opportunity of performing, in principle, an exhaustive examination of the program's code and of coming to an effective answer as what concerns the behavioural properties of interest. Some of those properties, usually formalised in a specification language, are common to all programs, such as deadlock freedom; others are relative to classes of programs, as the liveness property of reaching a desired state or avoiding an undesired state under some specified conditions (Baier and Katoen 2008, 12). In formal methods, programs' code is algorithmically checked against those specifications. The effective methods thereby provided are of particular significance in all those contexts in which empirically testing the system is either unfeasible (such as with software involved in robotic-aided surgery) or has unacceptably expensive costs (such as testing rocket controller software).

Since the original development of formal methods in the seventies, a debate arose between mathematicians and engineers as what concerns the actual reliability of formal methods in the evaluation of the correctness of programs with respect to the desired set of specifications (Shapiro 1997). In a famous paper, Hoare (1969) maintained that logic enables one to determine the set of allowed behaviours of a given program in terms of the closed set of consequences deduced within an axiomatic system representing the program. Since Hoare's original essay, Theorem Proving put faith in the opportunity of acquiring a-priori proofs of programs' correctness and thereby of defining *mathematical theories* of software systems. Such theories are syntactic theories characterised by a set of axioms, formalising enabling conditions for programs' executions, and by a set of rules of inference enabling one to deduce allowed computations from the specified set of axioms. Many objections to mathematical proofs of correctness appealed to the undecidability resulting from the incompleteness of those axiomatic systems, others on the complexity of carrying out proofs which therefore demand for computer aid, others on the difficulty of providing well-defined specifications (Shapiro 1997).

Fetzer (1988) provided a rather methodological objection to the feasibility, for formal methods, to provide predictions of programs' behaviours that are coherent with observed executions. Any formal method makes inferences on programs as abstract machines, that is, as mathematical entities about which it is unsurprisingly possible to perform deductive reasoning. However, one would like to evaluate the correctness of the physical machines instantiating those abstract machines. A mathematically correct program might produce incorrect executions when instantiated; this might be due to many reasons, including hardware exceptions or unexpected interactions with users and environments.

If one is involved in the predictions and explanations of actual executions of physical running machines, *empirical theories* are required. Angius and Tamburrini (2011) propose an account in this perspective introducing *semantic theories* (Suppe 1989) of software systems defined by a set of computational models representing *observed* executions of the verified program. Models are organised into an abstracting/instantiating hierarchy mapping executions of an abstract state transition system used in formal verification into paths of a *model of data* (Suppes 1962) representing observed executions. Mappings ensure that actual executions correspond to abstract paths in the state transition system, thus meeting Fetzer's objection.

Suppes (1962) suggested that, in a semantic theory, mappings from abstract theoretical models to concrete models of data cannot be given directly, but by means of a hierarchy of abstract mapped models, as depicted in Figure 1. At the top of such hierarchy, an abstract transition system enabling one to perform an algorithmic check of the model, such as by means of the Model Checking techniques (Baier and Katoen 2008). Subsequent concretizations of the theoretical model lead to the *model of the experiment*, containing paths representing those executions that ought to be observed in order to empirically evaluate the correctness of the program involved. At the bottom of the semantic hierarchy, a model of data concretizes models of experiments by containing instances of the runs represented in the latter and which should correspond, in case of adequate representation, to observed runs.

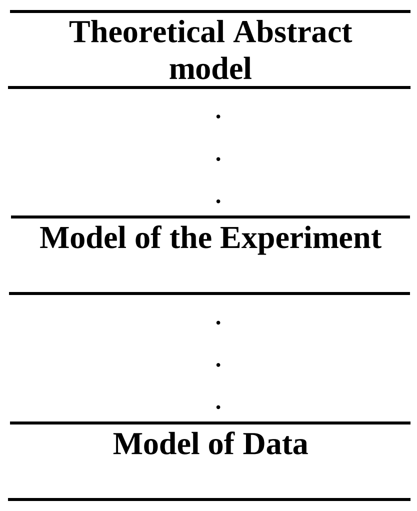

Fig.1 A semantic theory's representational hierarchy according to Suppes (1962).[1]

It should be noted here how computational models identifiable with theoretical models, models of the experiments, and of models of data, are of different types in the actual practice of computer science. Kripke structures and other labelled automata are mostly used in Model Checking, whereas data flow graphs are often utilised to model observed executions in software testing. A first difficulty in the construction of the semantic theories of software systems of Figure 1

[1] Suppes (1962, 259) considers even more levels at the bottom of the hierarchy and corresponding to the experimental design conditions and to *ceteris paribus* conditions. As being involved in the process of the experiment set-up and of data collection, they will not be taken into consideration here.

is providing mapping relations among structures modelled using different formalisms. Most importantly, providing a state transition system or a data flow graph representing all executions of a non-trivial program is hardly feasible. Both formal verification and testing techniques are carried out modularly, that is, the program's modules are evaluated independently (Müller 2002). Models used to represent different modules are also represented using different logics, even when performing the same verification technique but in case divergent properties are to be checked in each module.

This paper faces the problem of *discovery* of semantic theories of *modular* software systems, that is, of achieving the semantic hierarchy of Figure 1 from a collection of models, expressed within heterogeneous formalisms and representing different modules of the same program. Section 2 suggests that logic relations among models representing different program's modules be provided in such modular theories. Before proposing a structure for those modular theories, section 3 takes into consideration scientific structuralism (Balzer et al. 1987) to show how it is not able to represent heterogeneous modules interactions. Finally, section 4 introduces to the *Theory of Institutions* (Goguen and Burstall 1992), an abstract model theory applied to software specification languages, to construct empirical theories of modular computational systems.

2. Discovering Empirical Theories of Software Systems

In the context of the ongoing debate dividing verificationists from testers, the mathematician Goguen (1992) highlighted how an "error – free" top-down approach from abstract machine correctness to physical machine correctness cannot be pursued: whether the running computational system is a fair instantiation of the abstract state transition system cannot be a-priori settled. Indeed, computational models used in formal verification are usually built upon the program's available code and hardware correctness is usually assumed in the form of the so-called fairness constraints (Clarke et al. 1999, p. 32). By contrast, models of experiments and models of data represent failures that might be due to both program faults (bugs) and hardware exceptions.

We maintain here that a bottom-up approach is unattainable as well. State transition systems are used in formal verification to check whether desired behaviours belong to allowed behaviours; to do to this, the model is required to represent, in principle, all potential executions of the target program. This might even involve the representation of never-ending runs. On the contrary, models used to perform experiments on computational systems contain finite segments of those infinite paths and only a subset of runs needed to perform the required tests is represented. So it is not practicable to start with representations used in software testing to obtain state transition systems to be used to perform algorithmic verification.

Formal verification and testing are two distinct processes, in the evaluation of software systems, that are carried out independently. Code verification cannot be avoided in the construction of empirical theories of software systems: once a failure is observed by testing a system, what engendered that failure has to be identified, that is, testers are interested in *explanations* of the undesired observed behaviours. By only testing the system, it is not possible to determine whether observed failures are induced by hardware exceptions or faulty code lines[2]. The observation of incorrect runs of a program that is formally correct induces one to think that a problem arose with the implementation of the tested program; and in case of incorrect artefacts, formal methods enable

[2] Many debuggers simply face this problem by adding an exception handling in the tested program's code. Another strategy is to test the same program using a different implementation (Ammann and Offutt 2008, 231)

one to isolate the error state among the instruction lines. Following Fetzer (1988), testing cannot be avoided as well: programs that are formally correct might result in incorrect executions both in case the developed models are not adequate representations of the verified program, and in case the implementing hardware is flawed by design errors or physical failures.

A semantic theory of a software system is a structure systematizing and justifying the attained knowledge about a studied class of software systems. Indeed, this is the aim of any scientific theory, being it expressed in a syntactic or a semantic way. To reword Goguen's (1992) remark, such theories are conceived in a *context of justification* that does not resemble the way theories are discovered. Computer science is characterised by many modelling activities of computational systems at different levels of representation, including the state transition system level, the programming language level, the hardware architecture level, the logic circuit design level, etc..

By excluding purely deductive top-down approaches in software verification, Goguen (1996) suggested that in-formal methods must be somehow involved in the process of evaluations of software systems. According to his view "any formalism is situated" and "without human intervention, a formalization may well be inadequate for its intended applications". So, even the most well defined formal approach requires a context for its interpretation. Nice examples for the limits of formalization come from requirements engineering, where documents are often vague or even deliberately misleading, but this informality has real advantages. Vagueness and ambiguity help to describe tradeoffs that need a lot of work to be resolved, usually at a latter stage of the software life cycle.

We address the informality demanded by Goguen in the *abductive process* of hypothesising models, at different levels in the semantic hierarchy, and of refining them until they provide successful predictions (Angius 2013). In formal verification, state transition systems are hypotheses concerning all potential executions of the represented software system; they are conceived by taking into consideration the program's instructions. Algorithmic verification is afterwards performed on those models and against the desired set of property specifications. Some specifications might be positively verified while others might be violated. Those results are model-based hypotheses whose predictive adequacy still have to be settled by testing the system. In case observed executions are not consistent with those predictions and the former are correct runs, the involved model is refined.

Computational paths of some given state transition system which are incorrect with respect to a property specification, and to which correspond any of the actual program's executions, are often called false negatives; whereas false positives are correct paths not representing concrete computations. Model refinements aim at deleting false negatives and false positives that are recognised as such while testing the software system. Both of them are usually produced by an inadequate granularity of the transitions of the state transition system involved (Clarke et *al*. 1999, p. 16). Granularity is higher when transitions occur between macro states to which correspond many actual states of the represented software system. And granularity is lower when transitions take place between states to which no actual state, nor set of states, correspond; rather, an actual system's state corresponds to a set of states in the model. Actual states are actual assignments to the program's variables and transitions occur when those variables are assigned new values, consistently with the program's instructions. Inadequate models may be refined by decreasing or increasing the granularity of the transitions. For instance, a false negative may be generated by a high granularity of transitions. Suppose a model checker detect an incorrect path from, say, state s_n

to state s_m ($n, m \geq 0$; $n \neq m$); suppose also that to state s_n in the model correspond states $s_{n_1}$and s_{n_2} in the system, and to state s_m in the model correspond states $s_{m_1}$and s_{m_2} in the system. Finally suppose that two executions are observed, one from state $s_{n_1}$to state s_{n_2}, and another one from state $s_{m_1}$to s_{m_2}. Clearly, there is no actual execution corresponding to the modelled transition from state s_n to state s_m: the latter is a false negative which can be removed by decreasing the granularity of the transition, that is, by modelling two distinct model transitions, one from state $s_{n_1}$to state s_{n_2}, and another one from state $s_{m_1}$to s_{m_2} (Clarke et *al*. 2000).

The discovery of models constituting semantic theories of computational systems goes through a *trial and error process* involving several refinements and human interventions. Statements holding true in a refined model, that is, logic formulas expressing the verified property specifications, are those discovered law-like statements the model makes true (Angius and Tamburrini 2011).

The same can be said about models involved in software testing. Data flow graphs hypothesise a set of error states and incorrect paths the tester believes might be executed by the program to be evaluated (Ammann and Offutt 2008). The system is subsequently tested according to that model, i.e., the tester uses the model to select executions to be observed. In case a represented failure is not detected among performed runs, the model is refined accordingly, that is, the counterexample path is removed from the hypothesised failures. And in case executed faults are observed which are not represented by some counterexample in the data flow graph, any corresponding false positive is removed to be replaced by such counterexample.

It should be noted how, both while refining state transition systems or data flow graphs, for a counterexample to be recognised as a false negative, a problem arises about when to stop testing the system. Indeed, for an hypothesised counterexample, not being observed during the testing phase is not a guarantee that it will not be observed in the future. This problem follows the ‘Dijkstra’s dictum’: “Program testing can be used to show the presence of bugs, but never to show their absence” (Dijkstra 1970, p.7). Model-based hypotheses concerning future computations of a represented software system are tested up to a certain time fixed by testers; they are assigned a probabilistic evaluation according to statistical estimations of future incorrect executions based on past observed failures (Littlewood and Strigini 2000). Accordingly, they assume the epistemological status of probabilistic statements that are corroborated by failed attempts of falsification (Angius 2014) .

3. Modular Semantic Theories and Empirical Structuralism

Each module in a program can be represented by some state transition system including all the allowed transitions; and a semantic theory, in the form of the abstracting hierarchy described above, can be provided for each of those modules. Let us suppose to provide a semantic theory of a modular software system in terms of a set of representational hierarchies, each for any module in the program. Such theory would not be adequately predictive with respect to the target system’s behaviours, the reason being that modules are not independent entities. Modules usually interact with each other, bringing about new computations that are not performed by any module in isolation. From a logical viewpoint, module interactions can be grouped into refinement, integration, and composition relations (Diaconescu, Goguen and Stefaneas 1993). Informally, refinements are stepwise processes transforming an abstract – usually formal – specification into an executable lower level program.[3] Refinements follow logical rules of entailment from the one specification to the next more concrete one, which displays the same behaviour. ‘More concrete’, in this case, means be more effectively or more directly implemented. The composition (sum) of specifications is to “put together” two (or more) specifications usually written with the help of the same formal language and create a new specification. Hence, the most straightforward application of composition is the potential creation of “large” specifications from “smaller” ones. By integration of specifications we mean a more abstract form of composition which can be realised over more than one formal language.

Let us consider a simple example of a library, liberally taken from (Guerra 2001; Sernada et *al*. 1995). To build an initial specification for the library, some predicates are introduced, such as $available$, stating that a given item (a book) can be borrowed from the library; $taken$, expressing that the book has been borrowed; and $returned$, indicating that the book has been given back. These predicates allow to describe how the represented system (the library) ought to behave, in other words, they enable one to provide a specification, call it S, for the library. S can be expressed syntactically, by a set of statements $Sen(S)$, or semantically by a set of models $Mod(S)$ wherein those statements hold true. For instance, it should be required that only $available$ book can be $taken$ from the library; this can be expressed by Linear Time Temporal Logic (LTL) formula $\boldsymbol{G}(taken \rightarrow available)$. $\boldsymbol{G}$ is the temporal operator ‘Globally’ which, in this case, ensures that in any state (globally) of the transition system if $taken$ holds then $available$ must also hold. Another of such behavioural properties may be $\boldsymbol{G}(taken \rightarrow \boldsymbol{X} \neg available)$, stating that, for any state where $Taken$ holds, in the next ($\boldsymbol{X}$) state $available$ does not hold. This expresses that borrowed books are not available any more. But if a borrowed book is $returned$, it will be available again; formally: $\boldsymbol{G}(returned \rightarrow \boldsymbol{X} available)$.[4]

S can be refined by adding additional statements that render the description of the system more concrete; a refinement of S is a specialising specification S' which statements $sen(S') \supset sen(S)$ are a superset of the set of sentences of S. Books in a library can be reserved, making them un$available$ but not borrowed yet. Supplementary predicates can be added, such as *suspended* and *resumed*, respectively expressing that a book cannot be borrowed even if it has not been $taken$ and that the book is not reserved anymore. Refining statements include $\boldsymbol{G}(suspended \rightarrow \boldsymbol{X} \neg available)$ and $\boldsymbol{G}(suspended \rightarrow \boldsymbol{X}\,(\neg available \boldsymbol{U} resumed))$; the until operator $\boldsymbol{U}$ here requires that the book cannot be borrowed until it ceases to be reserved.[5] Notice that behaviours allowed by S are also allowed by S'.

In providing a set of specifications for a library, also users have to be considered, as well as their interactions with the system. In the object-oriented specification paradigm, specifications are obtained for each object; in the present case a specification for the user is required, call it T. Modelling and prescribing interactions between the library and the user signifies defining compositions between S (or S') and T (or integrations, in case the two specifications are expressed in different vocabularies). As predicates of T consider $takes$, $borrows$, and $returns$; some of the prescribing statements can be $\boldsymbol{G}(takes \rightarrow \boldsymbol{X}\, borrows)$, $\boldsymbol{G}(returns \rightarrow \boldsymbol{X} \neg\, borrows)$, and

[3] Module refinements involved in the implementations of abstract specifications should be carefully distinguished from model refinements examined in the previous section and dealing with the discovery of semantic theories.

[4] $\boldsymbol{G}((available \wedge \neg\, taken) \rightarrow \boldsymbol{X} available)$; $\boldsymbol{G}((\neg\, available \wedge \neg\, returned) \rightarrow \boldsymbol{X} \neg\, returned)$; and $\boldsymbol{G}(returned \rightarrow \neg\, available)$ might be additional specification statements.

[5] Clearly, it must also hold that $\boldsymbol{G}\neg(suspended \ \wedge\ taken\,)$ and $\boldsymbol{G}\neg(resumed \ \wedge\ returned\,)$.

$\boldsymbol{G}((borrows \wedge \neg\, returns\,) \rightarrow \boldsymbol{X}\, borrows)$.[6] Composing, say, S' with T means considering the interactions between behaviours imposed by $S^{'}$ and those imposed by T. For instance, when $takes$ holds in any state of the model implementing T, $taken$ must hold in the state transition system implementing $S^{'}$. The same should be said about $returns$ and $returned$. This shows how transitions of one model condition the transitions of the model of another module (in this case object) of the same software. This paper aims at specifying the logic relations holding among discovered models of different modules so that the semantic theory resulting from those models be adequately predictive with respect to the represented systems' executions.

Before that, it is worth asking whether the structuralist approach on scientific theories (Balzer et al. 1987) provides insights in the understanding of the modular semantic theories under consideration in this study. Indeed, in the structuralists program, empirical theories are semantically identified by the set of their models and a *theory-net* is introduced specifying intra-theoretical links among *theory-elements* in the net. A theory-element $T = \langle K, I \rangle$ is given by a so called theoretical core K and by the set of intended applications I. In a *theory-core* $K = \langle M_p, M, M_{pp}, C, L \rangle$, the class of potential models M_p is a class of structures satisfying a set of axioms with no empirical content and defining the theory's basic concepts. Actual models in the class $M \subseteq M_p$ are models that, besides satisfying axioms satisfied by M_p, also satisfy the theory's laws having empirical content and being expressed with concepts defined in potential models. Axioms defining concepts coming from external theories are satisfied by the class of partial potential models M_{pp}. Constraints in the class $C \subseteq Po(M_p)$, belonging to the power set of set of potential models, connect models of the same theory-element, whereas links in the class $L \subseteq M_p \times M_p^*$ correlate partial models of two different theory elements T and T^* into a theory-net. Finally, intended applications are model-theoretically understood as well, as being in the class $I \subseteq M_{pp}$ (Moulines 1996).

Let us now turn to ask whether the structuralist concepts of *constraints* and *links* can be used to grasp the notions of refinements, integrations, and compositions between models in a modular semantic theory. The case of classical particle mechanics (CPM) provided by Balzer et al. (1987, 103-108) is considered here. Potential models of CPM are models identified by a (non-empty) set of particles, a set of time points, a set of space points, and by a position function, a mass function, and a force function assigning space points, masses and force values respectively to particles in the set.[7] Actual models of CPM are models that, besides satisfying the axioms made true by potential models, satisfy Newton's second law. Among the potential models of a given theory, one might be interested in models meeting some given property of interest. In the case of CPM, one such property can be that given two applications of the theory, i.e. two mechanical systems (such as two planets), and a particles belonging to both systems (a grave moving from one planet to the other), the assigned mass is the same in the two systems. This constraint, known as the equality constraint, simplifies the calculation of a particle motion between the two systems and can be settled by requiring, for two potential models m and m° and a particle p, that $m(p) = m^{\circ}(p)$. C is thus the class of potential models satisfying the desired constraints (Balzer et al. 1987, 44-46).

[6] Other statements are $\boldsymbol{G}(takes \rightarrow \neg\, borows\,)$, $\boldsymbol{G}(returns \rightarrow borrows\,)$, and $\boldsymbol{G}((\neg\, borrows \wedge \neg\, takes\,) \rightarrow \boldsymbol{X} \neg\, borrows)$.

[7] To use the formalism of Balzer et al. (1987, 30), m is a potential model of classical particle mechanics iff i) $m = \langle P, T, S, c_1, c_2, s, m, f \rangle$; ii) P, T, S are non-empty sets of (finite) particles, time and space points respectively; iii) $c_1 : T \rightarrow \mathbb{R}$ and $c_2 : S \rightarrow \mathbb{R}^3$ are a time and a space bijective coordination function; iv) $s: P \times T \rightarrow S$ is the space function; v) $m: P \rightarrow \mathbb{R}^+$ is a mass function; vi) $f: P \times T \times N \rightarrow \mathbb{R}^3$ is a force function.

Potential models satisfy axioms defining all concepts used in the theory-element. Structuralists take into consideration the empirical positivism's distinction between theoretical and non-theoretical terms to isolate, among models in M_p, models in which only non-theoretical terms are involved.[8] Those terms are concepts coming from other, different, theory-elements and appear in partial potential models of the theory-element under consideration. Particles, time, and space are non-theoretical with respect to CPM. This means that they can be determined without the need of any actual model of CPM. Time and point space can be determined, for instance, by such theories as chronometry and physical geometry respectively (Balzer et al. 1987, 51-52). Partial potential models of CPM are thus models defining the particle, time, and space states together with the space function; mass and force function are omitted as being theoretical.[9] Non-theoretical terms in partial potential models of CPM are theoretical terms of partial models of some other theory-element. Intra-theoretical links in $L \subseteq M_p \times M_p$ allow one to use non-theoretical terms in a theory-element that are defined in a preceding theory-element.[10]

Let us ask whether constraints can define relations between models in a semantic hierarchy representing a program's module and whether intra-theoretical links are able to represent module interactions in a semantic modular theory. By considering the subset of models satisfying some property, constraints can be used to isolate all those structures that satisfy a given program specification. However, constraints do not represent logic relations, such as abstractions and refinements, among those models. A formalism able to map models at different levels of abstraction and of different types in order, for those models, to satisfy the same formulas, is still lacking.

As what concerns links, they are used to express specialization relations among theory-elements in a theory-net; links induce preorders among the actual models of specializing theory-elements; both potential and partial potential models are equal in all theory-elements.[11] Given a theory-element, further laws can be added to those satisfied by its actual models, thus restricting the domain of intended applications. Actual models of CPM satisfy Newton's second law; according to Balzer et al. (1987) a theory-element specialization of CPM is NCPM (Newtonian Classical Particle Mechanics) obtained by adding, to the actual models of CPM, Newton's third law concerning the *actio-reactio* principle. NCPM is a specialization of CPM since Newton's third law is less universal than his second law, that is, there are applications of CPM in which the third law is not required. Further specializations of CPM include Hooke classical particle mechanics HCPM, whose actual models also satisfy Hooke's law; gravitational classical particle mechanics GCPM, whose actual models satisfy the law of gravitation; and the electrostatic classical particle mechanics ECPM, involving Coulomb's law. More in general, theory-elements, satisfying a physical law that has general intended applications, induce a *tree-like* theory-net of specialised theory-elements by adding further laws that restrict the domains of application (Balzer et al. 1987, 175).

Links are specializing relations which can express refinements of modules, insofar as they induce a preorder on actual models. Adding axioms to subsequent models from CPM to NCPM, and

[8] The introduction of the distinction between theoretical and non-theoretical terms, and their respective models, is, according to Suppe (2000), a neo-postivistic heritage preventing scientific structuralism from being consistent with the semantic view of theories.

[9] The class M_{pp} in CPM is given by models $m_{pp} = \langle P, T, S, c_1, c_2, s \rangle$.

[10] For instance, some link must be established between $T$and $c_1$in partial potential models of CPP and the corresponding elements of potential models in chronometry theory.

[11] Given two theory-elements T and T', σ is a specialization relations $T \, \sigma \, T'$ iff $M_p = M'_p$; $M_{pp} = M'_{pp}$; $M \subseteq M'$; $C \subseteq C'$; $L \subseteq L'$; $I \subseteq I'$ (Moulines 1996, 11).

so on, is akin to considering further statements in specialising specifications from S to S', to S'', and so on. The first is a process of concretization of models, the second is a process of implementation of specifications. However, the more important relations of integration and composition are left out of this picture. Also, it is very unlikely that the preorder of models of a modular program displays a tree-like structure.[12]

Beyond all this, there is a main meta-theoretical and methodological reason at the base of the difference between the structuralists' aims and the present concern. The structuralist project pursues the objective of indentifying the structure of *existing* scientific theories, in terms of theory-nets consisting of theory-elements connected by links and constraints, and of underlining the relations of equivalence, specialization, and reduction among those theories. The far-reaching goal is that of reducing main theories to each other into a unificationsit, olsistic, picture. On the other hand, this study is involved in the discovery of new, *non-existing* theories. Specifically, the main objective is here that of getting from a collection of available computational models, representing different modules of a software system at several levels of abstraction, into a semantic theory of such system. Once one such modular semantic theory is available, the structuralist formalism might be used to reconnect theories of a class of programs into a theory-net.

4. Using Institutions to build Modular Semantic Theories

The Theory of Institutions is an abstract model theory applied to software specification languages. It was introduced by Goguen and Burstall (1992) to face the vast number of formalisms characterising common specification activities. Based on category theory (Goguen 1991), Institutions abstract from both the syntax and the semantics of a given language to focus on the satisfaction relation of models. Institutions accomplish the Tarskian satisfaction condition requiring that truth is invariant under change of notation (Tarski 1944). In contrast with Barwise's (1974) abstract model theory, Institutions apply to any-order language and to many-sorted logics. They are used to formalise programs' specifications by providing syntactic and semantic descriptions of the programs' modules. By using common categorical constructs, Institutions allow one to connect "small" specifications to obtain "larger" specifications (Burstall and Goguen 1977).

Informally, an Institution is introduced by indicating a collections of signatures, i.e. vocabularies, one would like to utilize in the description of a piece of software; a collection of sentences of interest which are expressed by using the defined vocabularies; the set of models, each expressed within a given signature and satisfying those sentences; and a satisfaction relation which be independent from the chosen signature. All these classes are defined categorically by means of an Institution $I = (\boldsymbol{Sign}, Sen, Mod, \vDash_{\Sigma})$ wherein $\boldsymbol{Sign}$ is a category of signatures; $Sen\colon \boldsymbol{Sign} \rightarrow Set$ is a functor, that is, a mapping of morphisms, defining the set of sentences expressible with each signature Σ in $\boldsymbol{Sign}$; another functor $Mod\colon \boldsymbol{Sign} \rightarrow \boldsymbol{Cat}^{op}$ introduces the category of models satisfying the defined formulas; $\vDash_{\Sigma} \subseteq |Mod(\Sigma)| \times Sen(\Sigma)$ is a Σ-satisfaction relation construed as follows: given a model and a formula satisfied by that model, a morphism in the category of models

[12] Structuralists maintain that global inter-theoretical relations are also given among theory-elements of different theory-nets, such as between models of CPM and models of collision mechanics (Balzer et al. 1987, ch. 6). This kind of links assumes the forms of specialization, reduction, equivalence, and approximation. Beside the fact that, again, these are not the types of relation needed in the construction of modular semantic theories, global inter-theoretical relations hold among different macro-theories, whereas the topic of concern here is the modular structure of each macro-theory.

maps from the model into the translation of the model in a different vocabulary in such a way that the translated model satisfies the translation of the formula (Goguen and Burstall 1992, 10). [13]

Institutions can be used to provide descriptions of modules in a program. Modules are represented by means of a so-called theory T over an Institution *I*. Once defined a given Institution, a theory $T = (\Sigma, E^{**})$ is introduced by choosing a profitable signature Σ and by formalising within Σ a set E^{**} of sentences describing the module and that be closed under entailment. *Galois connections* enable one to determine the closed set of models M^{**}satisfying the theory's sentences.[14]

The defined Institution possesses many vocabularies apart from Σ and it thus allows to describe the same "abstract module" within different formalisms, the requirement being that morphisms be given between sentences of each theory or, equivalently, between models satisfying those sentences. In other words, the same "abstract module" can be described by different theories selecting different signatures, provided that each sentence in a theory is the translation of a sentence in the other theory. A given Institution I induces a category **Th** of theories, which objects are theories of a specified program's module, and which morphisms, known as *theory morphisms*, express relations holding between translating theories. It is worth noting how theory morphisms in **Th** can express, besides translational equivalence of theories, also abstractions and refinements, i.e. logic relations formalising software engineering techniques on module specification and programming (Sanella and Tarlecki, 2012).

Objects and morphisms in the category **Th** of theories describing a given module can be used to define a semantic theory for that module. First, one can map, via Galois connections, a collection of corresponding models from the theories in the category. Note that each model is expressed with a different formalism in $\boldsymbol{Sign}$ (the signature of the corresponding theory); these models can be identified with the different typologies of models utilised in formal methods and testing in the evaluations of a program. Theory morphisms ensure that different models satisfy the same set of property specifications. Finally, one can consider data abstractions and refinements among models in an abstracting hierarchy by means of the opportune theory morphisms in **Th**.

Given a program encoded into several modules, one can preliminary represent each module by introducing an opportune Institution and indicating a category **Th** of theories over that institution. A modular semantic theory predicting the behaviours of the modular target system should include a series of Institutions, each providing a semantic theory for each program module, and a set of relations between models belonging to different theories. This can be formalised by considering the category **INS** of Institutions which objects are Institutions and which morphisms are the so-called *Institution morphisms* representing relations among Institutions (Goguen and Burstall 1992, 19-21). Institution morphisms are categorical constructs, such as colimits and pushouts, able to represent refinements, integrations, and compositions between couples of modules in a program (Diaconescu, Goguen, and Stefaneas 1993).

As already stated, integrations and compositions are the kinds of constructs of main interest in this study, insofar as they give rise to those unmodelled computations that should be predictable

[13] Formally, this is expressed by stating that for each morphism $\phi: \Sigma \rightarrow \Sigma'$ in $\boldsymbol{Sign}$, each signature Σ in $\boldsymbol{Sign}$, each $f \in Sen(\Sigma)$, and each $m \in |Mod(\phi)|$, $m \vDash Sen(\phi)(e)$ iff $Mod(\phi)(m) \vDash e$ (Goguen and Burstall 1992, 10).

[14] Galois connections establish, in Institutions Theory, a duality between model classes and theories by means of which any Σ-theory T determines the class of Σ-models $T°$ satisfying the Σ-sentences of T. And any Σ-model class V determines a Σ-theory $V°$ containing all the Σ-sentences satisfied by models in the class V.

by modular semantic theories. Both integrations and compositions of modules are represented by means of an *intermediary Institution*. In the former case, signatures, sentences, and models of the intermediary Institution are given by an opportune merging of the sentences, and models of the two integrating Institutions. Connections between sentences and models are obtained by considering appropriate Institution morphisms in **Ins** mapping from the models and the sentences of the integrating Institutions to the models and the sentences of the integrated Institution (Kuts et al. 2010, 44-45). In case of composition, signatures, sentences, and models of the composed intermediary Institution are mapped by chosen elements in the union sets of, respectively, the sets of signature, sentences, and models of the two composing Institutions (Kuts et. al 2010, 46-53).[15]

Consider the composition of specifications S' and T of the library exemplified in the previous section. An Institution $I^{S'}$for specification S' is given by a category of signatures among which one is used in S'; in this case LTL. The set of sentences include the LTL formulas expressed above and others that can be expressed using LTL to describe the library system. A theory over the obtained Institution is properly given by the chosen signature, i.e. LTL, and the chosen LTL sentences. Galois connections determine the class of models, from $Mod\left(I^{S'}\right)$, satisfying those sentences. The same holds for I^T. The composition of S' and T is achieved by means of an intermediary Institution, call it $I_T^{S'}$, representing a virtual unifying object having the library and the user as sub-objects. To consider the interactions between $takes$ and $taken$, and between $returns$ and $returned$, two predicates describing the composed object are to be introduced, that is, $take$ and $return$. The former holds in the composed model only when $takes$ and $taken$ hold in the composing models, and $return$ is true in a state of the composed model only when both $returns$ and $returned$ are true in the corresponding states of the starting models. Institution morphisms are established from $I_T^{S'}$ to S' and from $I_T^{S'}$ to T; concretely, $take$ is mapped to $takes$ and $taken$, $return$ to $returns$ and $returned$. Most interestingly for the present study, new LTL formulas can be expressed in $I_T^{S'}$ and that hold in the intermediary models, such as $\boldsymbol{G}\big(take \to \boldsymbol{X}(\neg\, available\boldsymbol{U}return)\big)$ and $\boldsymbol{G}(borrows \to \neg\, available)$.

The latter are statements describing computations coming from the interaction of the integrating or composing modules. Intermediary models should be included in a semantic modular theory of the whole software system. We propose to consider the category **Th** of theories over an intermediary Institution to build a semantic hierarchy for a modules' interaction. A model at the bottom of such hierarchy is a model of data representing the observed executions endangered by module interactions. A semantic modular theory can be given by the set of semantic hierarchies representing each module in the program and, for any two interacting modules, by a corresponding intermediary semantic hierarchy relating models of two interacting modules. To do this, we require that if a model, at any level of abstraction in a hierarchy representing a module, is related to a model of an intermediary hierarchy, the same relation should hold between models of the two hierarchies at any lower level of abstraction until models of data are obtained, as depicted in Figure 2. Categorically, this can be formalised by establishing proper functors mapping Institution morphisms defining intermediary models at a given level in the representational hierarchy and Institution morphisms defining intermediary models at the immediate lower level. A modular

[15] For a technical treatment of integrations and compositions the reader may refer to (Kuts et al. 2010).

semantic theory is, in this way, a net of theories related by intermediary models representing modules' interactions.

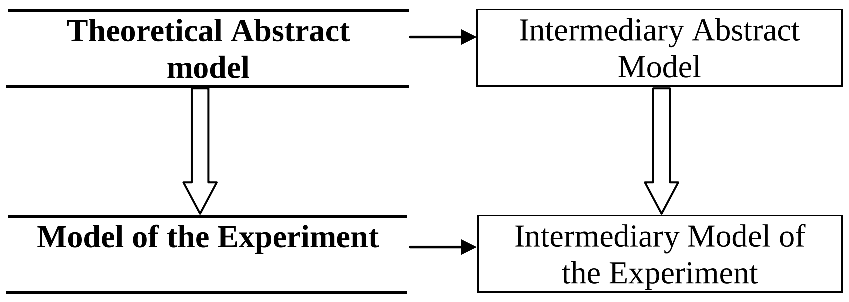
Theoretical Abstract model
Intermediary Abstract Model
Model of the Experiment
Intermediary Model of the Experiment

Fig. 2 Functors (bigger arrows) mapping Institution morphisms (smaller arrows) defining an intermediary abstract model and an intermediary model of the experiment. For the sake of simplicity, abstracting levels between the theoretical model and the model of the experiment have been omitted.

5. Concluding Remarks

Formal methods and software testing are involved in the hypothesis and the refinement of computational models, at different levels of abstraction, and for each module in the system to be evaluated. This paper is not involved in the improvement of any of those evaluation techniques. Rather, given the semantics of software verification and software testing, the philosophical aim is that of underlining what an empirical semantic theory of modular software system is.[16] By introducing an Institution and defining morphisms in the category **Th** of theories over that Institution, a semantic theory for each module in the program is provided. And by defining morphisms in the category **Ins** of module Institutions, a modular semantic theory representing interactions between couples of modules is constructed.

Interactions do not take place only *between* modules, but also *among* modules, that is, between group of modules. Executions arsing from the interactions of two modules may, in turn, interact with other executions. Institutionally, this can be formalised by considering integrations and compositions among intermediary Institutions, giving thereby rise to bigger intermediary Institutions. The process can be in principle reiterated until one gets to an Institution able to describe the whole of the modular program. Indeed, the original aim of applying abstract model theory to specification languages was that of computing the specification of a system starting from smaller specifications. Directly providing a specification of non-trivial programs is quite an hard task.

An Institution defining a specification for the whole software system can be considered to provide also, via the category **Th** of theories over that Institution, a semantic theory for the whole program in the traditional sense, that is, defined in terms of a single hierarchy of abstracting structures. Each model in this hierarchy is a model of the entire system, as in the former approach of Angius and Tamburrini (2011). We maintain that the process, conceived in the work of (Burstall

[16] The epistemological and methodological analysis advanced in this paper runs alongside with the technical attempts of drawing formal specifications from non-formal descriptions of modules to be reused in software development (Pandita et *al*. 2012).

and Goguen 1977; Goguen 1991; Goguen and Burstall 1992), of getting from module specifications to software specification, resembles the process of discovery of semantic theories of modular software systems. Depending on the kind of predictions and explanation one is seeking, and on the observed executions one would like to model, the modularity of the built semantic theory can be decreased by computing bigger Institutions describing a higher number of potential computations, until a non-modular semantic theory is achieved.

Acknowledgments
We are grateful to the three anonymous reviewers who advanced useful suggestions to improve the former draft of this paper.
This research has been co-financed by the European Union (European Social Fund- ESF) and Greek national funds through the Operational Program Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) – Research Funding Program: THALES: Reinforcement of the interdisciplinary and/or inter- institutional research and innovation.

References

Ammann, P., & Offutt, J. (2008). *Introduction to software testing*. Cambridge University Press.

Angius, N. (2013). Model-based abductive reasoning in automated software testing. *Logic Journal of the IGPL*, 21(6), 931–942.

Angius, N. (2014). The Problem of Justification of Empirical Hypotheses in Software Testing. *Philosophy and Technology*, 27(3), 423-439.

Angius, N., & Tamburrini, G. (2011). Scientific theories of computational systems in model checking. *Minds and Machines*, *21*(2), 323-336.

Baier, C., & Katoen, J. P. (2008). *Principles of model checking* (Vol. 26202649). Cambridge: MIT press.

Balzer, W., Moulines, C. U., & Sneed, J. D. (1987). *An architectonic for science: The structuralist program*. Dordrecht: Reidel.

Barwise, J. K. (1974). Axioms for abstract model theory. *Annals of Mathematical Logic*, *7*(2), 221-265.

Burstall, R. M., & Goguen, J. A. (1977). Putting theories together to make specifications. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2* (pp. 1045-1058). Morgan Kaufmann Publishers Inc.

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

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2000). Counterexample-guided Abstraction Refinement. *Proceedings of the 12th international conference for computer-aided verification. Lecture Notes in Computer Science*, 1855, 154–169.

Diaconescu, R., Goguen, J., & Stefaneas, P. (1993). Logical support for modularization. In *Second Annual Workshop on Logical Enviroments*, 83-100.

Dijkstra, E.W. (1970). Notes on structured programming. T. H.—Report 70-WSK-03.

Fetzer, J. H. (1988). Program verification: The very idea. *Communications of the ACM*, *31*(9), 1048-1063.

Fisher, M. (2011). An Introduction to Practical Formal Methods Using Temporal Logic. John Wiley & Sons.

Goguen, J. A. (1991). A Categorical Manifesto. *Mathematical Structures in Computer Science*, 1(1), 49-67.

Goguen, J. A. (1992). The denial of error. In C. Floyd, H. Züllighoven, R. Budde, & R. Keil-Slawik *Software Development and Reality Construction,* pp. 193-202. Berlin, Heidelberg: Springer.

Goguen, J. A. (1996). Formality and Informality in Requirements Engineering. *ICRE,* 96, 102-108.

Goguen, J. A., & Burstall, R. M. (1992). Institutions: Abstract model theory for specification and programming. *Journal of the ACM (JACM)*, *39*(1), 95-146.

Guerra, S. (2001). Composition of default specifications. *Journal of Logic and Computation*, *11*(4), 559-578.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, *12*(10), 576-580.

Littlewood, B., & Strigini, L. (2000). Software reliability and dependability: a roadmap. *ICSE '00 Proceedings of the Conference on the Future of Software Engineering*, 175–188.

Moulines, C. U. (1996). Structuralism: The Basic Ideas. In W. Balzer & C. U. Moulines (Eds.), *Structuralist theory of science: focal issues, new results,* pp. 1-21. Berlin: Walter de Gruyter.

Müller, P. (2002). *Modular Specification and Verification of Object Oriented Programs*. Berlin, Heidelberg: Springer-Verlag.

Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S., & Paradkar, A. (2012). Inferring method specifications from natural language API descriptions. *Proceedings of the 2012 International Conference on Software Engineering,* 815-825, IEEE Press.

Sanella, D., & Tarlecki, A. (2012). *Foundations of Algebraic Specifications and Formal Software Development*. Berlin, Heidelberg: Springer-Verlag.

Sernadas, A., Sernadas, C., & Costa, J. F. (1995). Object Specification Logic. *Journal of Logic and Computation*, 5, 603-630.

Shapiro, S. (1997). Splitting the difference: the historical necessity of synthesis in software engineering. *Annals of the History of Computing*, IEEE, 19(1), 20–54.

Suppe, F. (1989). *The semantic conception of theories and scientific realism*. University of Illinois Press.

Suppe, F. (2000). Understanding Scientific Theories: An Assessment of Developments, 1969, 1998. *Philosophy of Science,* 67, S102–S115

Suppes, P. (1962). Models of data. In E. Nagel, P. Suppes, & A. Tarski (Eds.), *Logic, methodology, and philosophy of science: Proceedings of the 1960 International congress* (pp. 252–261). Stanford University Press: Stanford.

Tarski, A. (1944). The semantic conception of truth: and the foundations of semantics. *Philosophy and phenomenological research*, *4*(3), 341-376.